

July 18–22, 2020  
Virtual Event, USA



Association for  
Computing Machinery

*Advancing Computing as a Science & Profession*



# ISSTA '20

Proceedings of the 29th ACM SIGSOFT International Symposium on

## Software Testing and Analysis

*Edited by:*

**Sarfraz Khurshid and Corina S. Păsăreanu**

*Sponsored by:*

**ACM SIGSOFT**

*Supported by:*

**Mooctest, DataVector AI**

Association for Computing Machinery, Inc.  
1601 Broadway, 10th Floor  
New York, NY 10019-7434  
USA

Copyright © 2020 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.  
Fax +1-212-869-0481 or E-mail [permissions@acm.org](mailto:permissions@acm.org).

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, USA.

#### **Notice to Authors of Past ACM-Published Articles**

ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that was previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform [permissions@acm.org](mailto:permissions@acm.org), stating the title of the work, the author(s), and where and when published.

ACM ISBN: 978-1-4503-8008-9

Additional copies may be ordered prepaid from:

ACM	Phone: 1-800-342-6626
General Post Office	(USA and Canada)
P.O. Box 30777	+1-212-626-0500
New York, NY 10087-0777	(All other countries)
	Fax: +1-212-944-1318
	E-mail: <a href="mailto:acmhelp@acm.org">acmhelp@acm.org</a>

Cover photo "Daytime view of Downtown L.A., as seen from Hollywood" by Thomas Pintaric / Licensed under CC BY-SA 3.0 / Cropped from original at <https://commons.wikimedia.org/wiki/File:LosAngeles05.jpg>

**Production:** Conference Publishing Consulting  
D-94034 Passau, Germany, [info@conference-publishing.com](mailto:info@conference-publishing.com)

## **Message from the Chairs**

We are delighted to welcome you to the ACM SIGSOFT International Symposium on Software Testing and Analysis, the 2020 edition (ISSTA 2020). ISSTA 2020 will be held as a virtual conference on July 18-22, 2020, and will be accompanied by two workshops, a doctoral symposium, a tool demo track, and a summer school.

ISSTA 2020 attracted a record number of submissions, ranging over a variety of topics related to software testing and analysis, with a noticeable increase in papers related to machine learning. In particular, ISSTA 2020 received 162 papers, out of which the program committee selected 43 high quality papers to appear in the conference. Each submission received at least three reviews. We thank our program committee members for their help in producing high quality reviews. ISSTA 2020 was the first ISSTA to use full double-blind reviewing where the author identities were not revealed until the decisions were final. Moreover, the identities for rejected papers were not revealed. Four of these papers have been selected for the ACM Distinguished Paper Award.

We are honored to have the ISSTA 2020 keynote by Barbara Liskov (MIT). Prof. Liskov is one of the first women to receive a doctorate in computer science in the United States. She is a winner of the 2008 Turing Award for her work on “programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing”.

ISSTA 2020 also presents two “test-of-time” awards. The ISSTA 2020 Impact Paper Award goes to the ISSTA 2010 paper entitled “Mutation-driven generation of unit tests and oracles” co-authored by Gordon Fraser and Andreas Zeller. The ISSTA 2020 Retrospective Impact Paper Award goes to the ISSTA 1994 paper entitled “Selecting tests and identifying test coverage requirements for modified software” co-authored by Gregg Rothermel and Mary Jean Harrold.

Like previous years, ISSTA 2020 had an artifact evaluation process, which assists authors of accepted papers provide more substantial supplements to their papers so that future researchers can more effectively build on and compare with previous work. A total of 20 artifacts were submitted. Each submission received three reviews from the artifact evaluation committee. 14 (out of the 20) submissions received the “Artifacts Evaluated - Functional” badge. Of these, 9 submissions also received the “Artifacts Evaluated - Reusable” badge.

The ISSTA 2020 tool demo track received 27 submissions. Each submission was reviewed by 3 members of the tool demo committee. 9 tool demo submissions were accepted to appear in the conference.

The doctoral symposium track received 5 submissions. Of these, 3 submissions were accepted by the doctoral symposium committee for presentation at the symposium.

ISSTA 2020 has two co-located workshops: the International Workshop on Smart Contract Analysis (WoSCA 2020), and the Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things (TAV-CPS/IoT). These workshops bring together researchers looking at cutting edge testing and analysis technologies in emerging domains of high importance.

The ISSTA 2020 summer school features a strong group of speakers from academia and industry. The summer school aims to encourage students to participate in research opportunities in the broad areas of testing and analysis.

We look forward to an interesting program at ISSTA 2020, and we extend our warmest thanks to all the authors, program committee members, attendees, and organizers whose contributions and engagement make this conference possible.

*Sarfraz Khurshid (General Chair)*

*Corina Păsăreanu (Program Chair)*

# ISSTA 2020 Organizing Committee

**Sarfraz Khurshid**, General Chair

University of Texas at Austin, United States

**Corina S. Păsăreanu**, Program Chair

Carnegie Mellon University Silicon Valley, NASA Ames Research Center, United States

**Milos Gligoric**, Organizing Chair

University of Texas at Austin, United States

**Oksana Tkachuk**, Workshops Co-Chair

Amazon Web Services, United States

**Lingming Zhang**, Workshops Co-Chair

University of Texas at Dallas, United States

**Rody Kersten**, Artifact Evaluation Co-Chair

Synopsys, United States

**Kasper Luckow**, Artifact Evaluation Co-Chair

Amazon Web Services, United States

**Marcel Böhme**, Doctoral Symposium Co-Chair

Monash University, Australia

**Jaco Geldenhuys**, Doctoral Symposium Co-Chair

University of Stellenbosch, South Africa

**Owolabi Legunsen**, Tool Demonstration Co-Chair

University of Illinois at Urbana-Champaign, United States

**Tingting Yu**, Tool Demonstration Co-Chair

University of Kentucky, United States

**Shin Hwei Tan**, Student Volunteer Co-Chair

Southern University of Science and Technology, China

**Mattia Fazzini** Student Volunteer Co-Chair

University of Minnesota, United States

**Yi Li**, Web Chair

Nanyang Technological University, Singapore

**Jonathan Bell**, Publicity Chair and Virtualization Chair

George Mason University, United States

**Ismet Burak Kadron**, Local Arrangements and Finance Chair

University of California at Santa Barbara, United States

**Wei Yang**, Sponsorship Chair

University of Texas at Dallas, United States

## ISSTA 2020 Program Committee

**Corina S. Păsăreanu** (PC Chair), CMU, NASA Ames, United States  
**Nazareno Aguirre**, University of Rio Cuarto, Argentina  
**Lucas Bang**, Harvey Mudd College, United States  
**Eric Bodden**, Heinz Nixdorf Institut, Paderborn U. and Fraunhofer IEM, Germany  
**Marcel Böhme**, Monash University, Australia  
**Tevfik Bultan**, University of California at Santa Barbara, United States  
**Maria Christakis**, MPI-SWS, Germany  
**Myra Cohen**, Iowa State University, United States  
**Marcelo D'Amorim**, Federal University of Pernambuco, Brazil  
**Antonio Filieri**, Imperial College London, United Kingdom  
**Gordon Fraser**, University of Passau, Germany  
**Milos Gligoric**, University of Texas at Austin, United States  
**Divya Gopinath**, NASA Ames (KBR Inc.), United States  
**William G.J. Halfond**, University of Southern California, United States  
**Dan Hao**, Peking University, China  
**Mark Harman**, Facebook and University College London, United Kingdom  
**Guy Katz**, Hebrew University of Jerusalem, Israel  
**Sarfraz Khurshid**, University of Texas at Austin, United States  
**Daniel Kroening**, University of Oxford, United Kingdom  
**Wei Le**, Iowa State University, United States  
**Xuan-Bach D. Le**, University of Melbourne, Australia  
**Darko Marinov**, University of Illinois at Urbana-Champaign, United States  
**Anastasia Mavridou**, NASA Ames Research Center / KBR, United States  
**Ali Mesbah**, University of British Columbia, Canada  
**Vijayaraghavan Murali**, Facebook, United States  
**Alex Orso**, Georgia Institute of Technology, United States  
**Ajitha Rajan**, University of Edinburgh, United Kingdom  
**Baishakhi Ray**, Columbia University, United States  
**Nico Rosner**, Amazon Web Services, United States  
**Koushik Sen**, University of California at Berkeley, United States  
**Elena Sherman**, Boise State University, United States  
**Junaid Haroon Siddiqui**, Lahore University of Management Sciences, Pakistan  
**Kathryn Stolee**, North Carolina State University, United States  
**Frank Tip**, Northeastern University, United States  
**Yves Le Traon**, University of Luxembourg, Luxembourg  
**Willem Visser**, Stellenbosch University, South Africa  
**Guowei Yang**, Texas State University, United States  
**Tingting Yu**, University of Kentucky, United States  
**Andreas Zeller**, CISPA Helmholtz Center for Information Security, Germany  
**Jie M. Zhang**, University College London, United Kingdom  
**Lingming Zhang**, University of Texas at Dallas, United States  
**Xiangyu Zhang**, Purdue University, United States  
**Thomas Zimmermann**, Microsoft Research, United States

## **ISSTA 2020 Workshops Committee**

**Oksana Tkachuk** (Co-Chair), Amazon Web Services, United States

**Lingming Zhang** (Co-Chair), University of Texas at Dallas, United States

### **International Workshop on Smart Contract Analysis (WoSCA)**

**Josselin Feist** (Co-chair), Trail of Bits, United States

**Gustavo Grieco** (Co-chair), Trail of Bits, United States

**Alex Groce** (Co-chair) Northern Arizona University, United States

**Elvira Albert**, Complutense University of Madrid, Spain

**Diego Garberovetsky**, University of Buenos Aires, Argentina

**Neville Grech**, University of Athens, Greece

**Bo Jiang**, Beihang University, China

**Shuvendu K. Lahiri**, Microsoft Research, United States

**Hakjoo Oh**, Korea University, South Korea

**Grigore Rosu**, University of Illinois at Urbana-Champaign, United States

**Valentin Wüstholtz**, ConsenSys Diligence, Germany

### **Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things (TAV-CPS/IoT)**

**Yan Cai** (Co-chair), Institute of Software, Chinese Academy of Sciences, China

**Tingting Yu** (Co-chair), University of Kentucky, United States

**Shaukat Ali**, Simula Research Lab, Norway

**Lei Bu**, Nanjing University, China

**Thao Dang**, CNRS/VERIMAG, France

**Alex Groce**, Northern Arizona University, United States

**BaekGyu Kim**, Toyota Motor North America (TMNA) R&D, United States

**Oleg Sokolsky**, University of Pennsylvania, United States

**Valerio Terragni**, Università della Svizzera Italiana, Switzerland

**Hong-Linh Truong**, Aalto University, Finland

**Rui Wang**, Capital Normal University, China

## ISSTA 2020 Tool Demo Committee

**Owolabi Legunsen** (Co-Chair), University of Illinois at Urbana-Champaign, United States  
**Tingting Yu** (Co-Chair) University of Kentucky, United States  
**Aldeida Aleti**, Monash University, Australia  
**Saba Alimadadi**, Simon Fraser University, Canada  
**Hamid Bagheri**, University of Nebraska-Lincoln, United States  
**Ahmet Celik**, Facebook, United States  
**Junjie Chen**, Tianjin University, China  
**Jürgen Cito**, MIT, United States  
**Hyunsook Do**, University of North Texas, United States  
**Mattia Fazzini**, University of Minnesota, United States  
**Yang Feng**, Nanjing University, China  
**Sonal Mahajan**, Fujitsu Labs of America, United States  
**Breno Miranda**, Federal University of Pernambuco, Brazil  
**Hoan Anh Nguyen**, Amazon, United States  
**John-Paul Ore**, North Carolina State University, United States  
**Hila Peleg**, University of California at San Diego, United States  
**Justyna Petke**, University College London, United Kingdom  
**Sabrina Souto**, State University of Paraiba - UEPB, Brazil  
**Michele Tufano**, Microsoft, United States  
**Weihang Wang**, University at Buffalo, SUNY, United States  
**Xiaoyin Wang**, University of Texas at San Antonio, United States  
**Ying Wang**, Northeastern University, China  
**Shiyi Wei**, University of Texas at Dallas, United States  
**Xusheng Xiao**, Case Western Reserve University, United States  
**Jifeng Xuan**, Wuhan University, China  
**Wei Yang**, University of Texas at Dallas, United States  
**Hongyu Zhang**, University of Newcastle, Australia

## ISSTA 2020 Artifact Evaluation Committee

**Rody Kersten** (Co-Chair), Synopsys, United States  
**Kasper Luckow** (Co-Chair), Amazon Web Services, United States  
**Ellen Arteca**, Northeastern University, United States  
**Cyrille Artho**, KTH Royal Institute of Technology, Sweden  
**Tegan Brennan**, University of California at Santa Barbara, United States  
**Thomas Bøgholm**, Aalborg University, Denmark  
**Maxime Cordy**, SnT, University of Luxembourg, Luxembourg  
**Renzo Degiovanni**, SnT, University of Luxembourg, Luxembourg  
**Aymeric Fromherz**, Carnegie Mellon University, United States  
**Bernard van Gastel**, Open University of the Netherlands, Netherlands  
**Jiaping Gui**, NEC Labs America, United States  
**Liana Hadarean**, Amazon, United States  
**Malte Isberner**, StackRox, Germany  
**Rahul Krishna**, Columbia University, New York, United States  
**Rami Gökhan Kici**, University of California at San Diego, United States  
**Breno Miranda**, Federal University of Pernambuco, Brazil  
**Malte Mues**, Technical University Dortmund, Germany  
**Yannic Noller**, Humboldt-Universität zu Berlin, Germany  
**Chao Peng**, University of Edinburgh, United Kingdom  
**Van-Thuan Pham**, Monash University, Australia  
**Pablo Ponzio**, University of Rio Cuarto, Argentina  
**Huascar Sanchez**, SRI International, United States  
**Tushar Sharma**, University of Wisconsin - Madison, United States  
**Vaibhav Sharma**, Amazon Web Services, United States  
**Andreas Stahlbauer**, University of Passau, Germany  
**Michael Tautschnig**, Amazon Web Services, United States  
**Leopoldo Teixeira**, Federal University of Pernambuco, Brazil  
**Alexi Turcotte**, Northeastern University, United States  
**Qi Xin**, Georgia Institute of Technology, United States  
**Zhiqiang Zang**, University of Texas at Austin, United States

## **ISSTA 2020 Doctoral Symposium Committee**

**Marcel Böhme** (Co-Chair), Monash University, Australia  
**Jaco Geldenhuys** (Co-Chair), University of Stellenbosch, South Africa  
**Aldeida Aleti**, Monash University, Australia  
**Antonia Bertolino**, CNR-ISTI, Italy  
**Sang Kil Cha**, KAIST, South Korea  
**Sudipta Chattopadhyay**, Singapore University of Technology and Design, Singapore  
**Marsha Chechik**, University of Toronto, Canada  
**Antonio Filieri**, Imperial College London, United Kingdom  
**Gordon Fraser**, University of Passau, Germany  
**Julia Lawall**, INRIA, France  
**Andreas Zeller**, CISPA Helmholtz Center for Information Security, Germany

# Contents

## Frontmatter

---

Message from the Chairs . . . . .	iii
ISSTA 2020 Organization . . . . .	v

## Research Papers

---

### Fuzzing

#### WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats

Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa — *Sapienza University of Rome, Italy* . . . . . 1

#### Active Fuzzing for Testing and Securing Cyber-Physical Systems

Yuqi Chen, Bohan Xuan, Christopher M. Poskitt, Jun Sun, and Fan Zhang — *Singapore Management University, Singapore; Zhejiang University, China; Zhejiang Lab, China; Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China* . . . . . 14

#### Learning Input Tokens for Effective Fuzzing

Björn Mathis, Rahul Gopinath, and Andreas Zeller — *CISPA, Germany* . . . . . 27

### Symbolic Execution and Constraint Solving

#### Fast Bit-Vector Satisfiability

Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang — *Hong Kong University of Science and Technology, China* . . . . . 38

#### Relocatable Addressing Model for Symbolic Execution

David Trabish and Noam Rinetzky — *Tel Aviv University, Israel* . . . . . 51

#### Running Symbolic Execution Forever

Frank Busse, Martin Nowack, and Cristian Cadar — *Imperial College London, UK* . . . . . 63

### Repair and Debug

#### Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach

Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang — *Peking University, China; University of Texas at Dallas, USA; Ant Financial Services, China* . . . . . 75

#### Automated Repair of Feature Interaction Failures in Automated Driving Systems

Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter — *University of Luxembourg, Luxembourg; Delft University of Technology, Netherlands; University of Ottawa, Canada; IEE, Luxembourg* . . . . . 88

#### CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan — *University of Waterloo, Canada; Purdue University, USA* . . . . . 101

### Mobile Apps

#### Detecting and Diagnosing Energy Issues for Mobile Applications

Xueliang Li, Yuming Yang, Yepang Liu, John P. Gallagher, and Kaishun Wu — *Shenzhen University, China; Southern University of Science and Technology, China; Roskilde University, Denmark; IMDEA Software Institute, Spain* . . . . . 115

#### Automated Classification of Actions in Bug Reports of Mobile Apps

Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang — *Beijing Institute of Technology, China* . . . . . 128

#### Data Loss Detector: Automatically Revealing Data Loss Bugs in Android Apps

Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani — *University of Milano-Bicocca, Italy* . . . . . 141

### Machine Learning I

#### Reinforcement Learning Based Curiosity-Driven Testing of Android Applications

Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li — *Nanjing University, China* . . . . . 153

#### Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy

Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh — *Korea University, South Korea* . . . . . 165

<b>DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks</b>	Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen – <i>Nanjing University, China; Hong Kong University of Science and Technology, China; Ant Financial Services, China</i>	177
<b>Machine Learning II</b>		
<b>Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries</b>	Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi – <i>University of Colorado Boulder, USA; TU Vienna, Austria</i>	189
<b>Higher Income, Larger Loan? Monotonicity Testing of Machine Learning Models</b>	Arnab Sharma and Heike Wehrheim – <i>University of Paderborn, Germany</i>	200
<b>Detecting Flaky Tests in Probabilistic and Machine Learning Applications</b>	Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic – <i>University of Illinois at Urbana-Champaign, USA</i>	211
<b>Bug Localization and Test Isolation</b>		
<b>Scaffle: Bug Localization on Millions of Files</b>	Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra – <i>University of Stuttgart, Germany; Facebook, USA</i>	225
<b>Abstracting Failure-Inducing Inputs</b>	Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller – <i>CISPA, Germany</i>	237
<b>Debugging the Performance of Maven’s Test Isolation: Experience Report</b>	Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric – <i>University of Texas at Austin, USA; Facebook, USA; George Mason University, USA; Microsoft, USA</i>	249
<b>Security</b>		
<b>Feedback-Driven Side-Channel Analysis for Networked Applications</b>	İsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan – <i>University of California at Santa Barbara, USA</i>	260
<b>Scalable Analysis of Interaction Threats in IoT Systems</b>	Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri – <i>University of Nebraska-Lincoln, USA</i>	272
<b>DeepSQLi: Deep Semantic Learning for Testing SQL Injection</b>	Muyang Liu, Ke Li, and Tao Chen – <i>University of Electronic Science and Technology of China, China; University of Exeter, UK; Loughborough University, UK</i>	286
<b>Regression Testing</b>		
<b>Dependent-Test-Aware Regression Testing Techniques</b>	Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie – <i>University of Illinois at Urbana-Champaign, USA; Google, USA; University of Washington, USA; Peking University, China</i>	298
<b>Differential Regression Testing for REST APIs</b>	Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk – <i>Microsoft Research, USA; University of Stuttgart, Germany</i>	312
<b>Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization</b>	Qianyang Peng, August Shi, and Lingming Zhang – <i>University of Illinois at Urbana-Champaign, USA; University of Texas at Dallas, USA</i>	324
<b>Challenging Domains</b>		
<b>Intermittently Failing Tests in the Embedded Systems Domain</b>	Per Erik Strandberg, Thomas J. Ostrand, Elaine J. Weyuker, Wasif Afzal, and Daniel Sundmark – <i>Westermo Network Technologies, Sweden; Mälardalen University, Sweden; University of Central Florida, USA</i>	337
<b>Feasible and Stressful Trajectory Generation for Mobile Robots</b>	Carl Hildebrandt, Sebastian Elbaum, Nicola Bezzo, and Matthew B. Dwyer – <i>University of Virginia, USA</i>	349
<b>Detecting Cache-Related Bugs in Spark Applications</b>	Hui Li, Dong Wang, Tianze Huang, Yu Gao, Wensheng Dou, Lijie Xu, Wei Wang, Jun Wei, and Hua Zhong – <i>Institute of Software at Chinese Academy of Sciences, China; University of Chinese Academy of Sciences, China; Beijing University of Posts and Telecommunications, China</i>	363
<b>Binary Analysis</b>		
<b>Patch Based Vulnerability Matching for Binary Programs</b>	Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu – <i>Xi'an Jiaotong University, China; Nanyang Technological University, Singapore; Fudan University, China; ShanghaiTech University, China; Zhejiang University, China</i>	376

<b>Identifying Java Calls in Native Code via Binary Scanning</b>	388
George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis — University of Athens, Greece . . . . .	
<b>An Empirical Study on ARM Disassembly Tools</b>	401
Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren — Hong Kong Polytechnic University, China; Zhejiang University, China; Arizona State University, USA; Nanyang Technological University, Singapore . . . . .	

## Static Analysis and Search-Based Testing

<b>How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools using Bug Injection</b>	415
Asem Ghaleb and Karthik Pattabiraman — University of British Columbia, Canada . . . . .	
<b>A Programming Model for Semi-implicit Parallelization of Static Analyses</b>	428
Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini — TU Darmstadt, Germany; KTH, Sweden . . . . .	
<b>Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing</b>	440
Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong — National University of Singapore, Singapore; Singapore Management University, Singapore; University of Passau, Germany; Xi'an Jiaotong University, China . . . . .	

## Build Testing

<b>Scalable Build Service System with Smart Scheduling Service</b>	452
Kaiyuan Wang, Greg Tener, Vijay Gullapalli, Xin Huang, Ahmed Gad, and Daniel Rall — Google, USA . . . . .	
<b>Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph</b>	463
Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang — Hong Kong University of Science and Technology, China; Xiamen University, China; Sourcebrella, China . . . . .	
<b>How Far We Have Come: Testing Decompilation Correctness of C Decompilers</b>	475
Zhibo Liu and Shuai Wang — Hong Kong University of Science and Technology, China . . . . .	

## Numerical Software Analysis and Clone Detection

<b>Discovering Discrepancies in Numerical Libraries</b>	488
Jackson Vanover, Xuan Deng, and Cindy Rubio-González — University of California at Davis, USA . . . . .	
<b>Testing High Performance Numerical Simulation Programs: Experience, Lessons Learned, and Open Issues</b>	502
Xiao He, Xingwei Wang, Jia Shi, and Yi Liu — University of Science and Technology Beijing, China; CNCERT/CC, China . . . . .	
<b>Functional Code Clone Detection with Syntax and Semantics Fusion Learning</b>	516
Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi — Nanjing University, China; Texas A&M University, USA; Hong Kong University of Science and Technology, China . . . . .	
<b>Learning to Detect Table Clones in Spreadsheets</b>	528
Yakun Zhang, Wensheng Dou, Jiaxin Zhu, Liang Xu, Zhiyong Zhou, Jun Wei, Dan Ye, and Bo Yang — Institute of Software at Chinese Academy of Sciences, China; Jinling Institute of Technology, China; North China University of Technology, China . . . . .	

## Tool Demonstrations

---

<b>ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity</b>	541
Ali Ghanbari — University of Texas at Dallas, USA . . . . .	
<b>Crowdsourced Requirements Generation for Automatic Testing via Knowledge Graph</b>	545
Chao Guo, Tieke He, Wei Yuan, Yue Guo, and Rui Hao — Nanjing University, China . . . . .	
<b>TauJud: Test Augmentation of Machine Learning in Judicial Documents</b>	549
Zichen Guo, Jiawei Liu, Tieke He, Zhuoyang Li, and Peitian Zhangzhu — Nanjing University, China . . . . .	
<b>EShield: Protect Smart Contracts against Reverse Engineering</b>	553
Wentian Yan, Jianbo Gao, Zhenhao Wu, Yue Li, Zhi Guan, Qingshan Li, and Zhong Chen — Peking University, China; Boya Blockchain, China . . . . .	
<b>Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts</b>	557
Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce — Trail of Bits, USA; Northern Arizona University, USA . . . . .	
<b>ProFL: A Fault Localization Framework for Prolog</b>	561
George Thompson and Allison K. Sullivan — North Carolina A&T State University, USA; University of Texas at Arlington, USA . . . . .	
<b>FineLock: Automatically Refactoring Coarse-Grained Locks into Fine-Grained Locks</b>	565
Yang Zhang, Shuai Shao, Juan Zhai, and Shiqing Ma — Hebei University of Science and Technology, China; Rutgers University, USA	

<b>CPSDebug: A Tool for Explanation of Failures in Cyber-Physical Systems</b>	
Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, Dejan Ničković, and Fabrizio Pastore — <i>TU Vienna, Austria; Austrian Institute of Technology, Austria; University of Milano-Bicocca, Italy; University of Luxembourg, Luxembourg</i>	569
<b>Test Recommendation System Based on Slicing Coverage Filtering</b>	
Ruixiang Qian, Yuan Zhao, Duo Men, Yang Feng, Qingkai Shi, Yong Huang, and Zhenyu Chen — <i>Nanjing University, China; Hong Kong University of Science and Technology, China; Mooltest, China</i>	573

## Doctoral Symposium

---

<b>Automated Mobile Apps Testing from Visual Perspective</b>	
Feng Xue — <i>Northwestern Polytechnical University, China</i>	577
<b>Program-Aware Fuzzing for MQTT Applications</b>	
Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista — <i>University of São Paulo, Brazil</i>	582
<b>Automatic Support for the Identification of Infeasible Testing Requirements</b>	
João Choma Neto — <i>University of São Paulo, Brazil</i>	587
<b>Author Index</b>	592

# WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats

Andrea Fioraldi

Sapienza University of Rome  
Italy  
andrea.fioraldi@gmail.com

Daniele Cono D'Elia

Sapienza University of Rome  
Italy  
delia@diag.uniroma1.it

Emilio Coppa

Sapienza University of Rome  
Italy  
coppa@diag.uniroma1.it

## ABSTRACT

Fuzzing technologies have evolved at a fast pace in recent years, revealing bugs in programs with ever increasing depth and speed. Applications working with complex formats are however more difficult to take on, as inputs need to meet certain format-specific characteristics to get through the initial parsing stage and reach deeper behaviors of the program.

Unlike prior proposals based on manually written format specifications, we propose a technique to automatically generate and mutate inputs for unknown chunk-based binary formats. We identify dependencies between input bytes and comparison instructions, and use them to assign tags that characterize the processing logic of the program. Tags become the building block for structure-aware mutations involving chunks and fields of the input.

Our technique can perform comparably to structure-aware fuzzing proposals that require human assistance. Our prototype implementation WEIZZ revealed 16 unknown bugs in widely used programs.

## CCS CONCEPTS

• Software and its engineering → Software testing and debugging; Software verification and validation; • Security and privacy → Software and application security.

## KEYWORDS

Fuzzing, binary testing, chunk-based formats, structural mutations

### ACM Reference Format:

Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397372>

## 1 INTRODUCTION

Recent years have witnessed a spike of activity in the development of efficient techniques for fuzz testing, also known as fuzzing. In particular, the coverage-based grey-box fuzzing (CGF) approach has proven to be very effective for finding bugs often indicative of weaknesses from a security standpoint. The availability of the AFL

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397372>

fuzzing framework [36] paved the way to a large body of works proposing not only more efficient implementations, but also new techniques to deal with common fuzzing roadblocks represented by magic numbers and checksum tests in programs. Nonetheless, there are still several popular application scenarios that make hard cases even for the most advanced CGF fuzzers.

CGF fuzzers operate by mutating inputs at the granularity of their bit and byte representations, deeming mutated inputs interesting when they lead execution to explore new program portions. While this approach works very well for compact and unstructured inputs [30], it can lose efficacy for highly structured inputs that must conform to some grammar or other type of specification.

Intuitively, “undisciplined” mutations make a fuzzer spend important time in generating many inputs that the initial parsing stages of some program typically rejects, resulting in little-to-none code coverage improvement. Researchers thus have added a user-supplied specification to the picture to produce and prioritize meaningful inputs: enhanced CGF embodiments of this kind are available for both grammar-based [4, 30] and chunk-based [24] formats, with the latter seeming prevalent among real-world software [24].

The shortcomings of this approach are readily apparent. Applications typically do not come with format specifications suitable to this end. Asking users to write one is at odds with a driving factor behind the success of CGF fuzzers, that is, they work with a minimum amount of prior knowledge [5]. Such a request can be expensive, and hardly applies to security contexts where users deal with proprietary or undocumented formats [8]. The second limitation is that testing only inputs perfectly adhering to the specification would miss imprecisions in the implementation, while inputs that are to some degree outside it may instead exercise them [8].

**Our approach.** The main feature of our proposal can be summarized as: *we attempt to learn how some chunk-based input structure may look like based on how a program handles its bytes*.

Nuances of this idea are present in [8] where code coverage guides grammar structure inference, and to some extent in a few general-purpose fuzzing additions. For instance, taint tracking can reveal which input bytes take part in comparisons with magic sequences [10, 25], while input-to-state relationships [5] can identify also checksum fields using values observed as comparison operands.

We build on the intuition that among comparisons operating on input-derived data, we can deem some as tightly coupled to a single portion of the input structure. We also experimentally observed that the order in which a program exercises these checks can reveal structural details such as the location for the type specifier of a chunk. We tag input bytes with the most representative comparison for their processing, and heuristically infer plausible boundaries for chunks and fields. With this automatic pipeline, we can apply

structure-aware mutations for chunk-based grey-box fuzzing [24] without the need for a user-supplied format specification.

We start by determining candidate instructions that we can consider relevant with respect to an input byte. Instead of resorting to taint tracking, we flip bits in every input byte, execute the program, and build a dependency vector for operands at comparison instruction sites. We analyze dependencies to identify input-to-state relationships and roadblocks, and to assign tags to input bytes. Tag assignment builds on spatial and temporal properties, as a program typically uses distinct instructions in the form of comparisons to parse distinct items. To break ties we prioritize older instructions as format validation normally happens early in the execution. Tags drive the inference process of an approximate chunk-based structure of the input, enabling subsequent structure-aware mutations.

Unlike reverse engineering scenarios for reconstructing specifications of input formats [12, 18, 19], experimental results suggest that in this context an inference process does not have to be fully accurate: fuzzing can deal with some amount of noise and imprecision. Another important consideration is that a specification does not prescribe how its implementation should look like. Developers can share code among functionalities, introducing subtle bugs: we have observed this phenomenon for instance in applications that operate on multiple input formats. Also, they can devise optimized variants of one functionality, specialized on e.g. value ranges of some relevant input field. As we assign tags and attempt inference over one input instance at a time, we have a chance to identify and exercise such patterns when aiming for code coverage improvements.

**Contributions.** By using information that is within immediate reach of a fuzzer in its usual work, we propose a method to infer an approximate structure for a chunk-based input and then mutate it in a fully automatic manner. Throughout the paper we present:

- a dependency identification technique embedded in the deterministic mutation stage of grey-box fuzzing;
- a tag assignment mechanism that uses such dependencies to overcome fuzzing roadblocks and to back a structure inference scheme for chunk-based formats;
- an implementation of the approach called **WEIZZ**.

In our experiments **WEIZZ** beats or matches a chunk-based CGF proposal that requires a format specification, and outperforms several general-purpose fuzzers over the applications we considered. We make **WEIZZ** available as open source at:

<https://github.com/andreaifioraldi/weizz-fuzzer>

## 2 STATE OF THE ART

The last years have seen a large body of fuzzing-related works [37]. Grey-box fuzzers have gradually replaced initial *black-box* fuzzing proposals where mutation decisions happen without taking into account their impact on the execution path [29]. Coverage-based *grey-box* fuzzing (CGF) uses lightweight instrumentation to measure code coverage, which discriminates whether mutated inputs are sufficiently “interesting” by looking at control-flow changes.

CGF is very effective in discovering bugs in real software [24], but may struggle in the presence of roadblocks (like magic numbers and checksums) [5], or when mutating structured grammar-based [8] or chunk-based [24] input formats. In this section we will describe the workings of the popular CGF engine that **WEIZZ** builds upon,

and recent proposals that cope with the challenges listed above. We conclude by discussing where **WEIZZ** fits in this landscape.

### 2.1 Coverage-Based Fuzzing

American Fuzzy Lop (AFL) [36] is a very well-known CGF fuzzer and several research works have built on it to improve its effectiveness. To build new inputs, AFL draws from a queue of initial user-supplied *seeds* and previously generated inputs, and runs two mutation stages. Both the *deterministic* and the *havoc* nondeterministic stage look for coverage improvements and crashes. AFL adds to the queue inputs that improve the coverage of the program, and reports crashing inputs to users as proofs for bugs.

**Coverage.** AFL adds instrumentation to a binary program to intercept when a branch is hit during the execution. To efficiently track *hit counts*, it uses a coverage map of branches indexed by a hash of its source and destination basic block addresses. AFL deems an input *interesting* when the execution path reaches a branch that yields a previously unseen hit count. To limit the number of monitored inputs and discard likely similar executions, hit counts undergo a normalization step (power-of-two buckets) for lookup.

**Mutations.** The deterministic stage of AFL sequentially scans the input, applying for each position a set of mutations such as bit or byte flipping, arithmetic increments and decrements, substitution with common constants (e.g., 0, -1, MAX\\_INT) or values from a user-supplied *dictionary*. AFL tests each mutation in isolation by executing the program on the derived input and inspecting the coverage, then it reverts the change and moves to another.

The havoc stage of AFL applies a nondeterministic sequence (or *stack*) of mutations before attempting execution. Mutations come in a random number (1 to 256) and consists in flips, increments, decrements, etc. at a random position in the input.

### 2.2 Roadblocks

*Roadblocks* are comparison patterns over the input that are intrinsically hard to overcome through blind mutations: the two most common embodiments are *magic numbers*, often found for instance in header fields, and *checksums*, typically used to verify data integrity. Format-specific dictionaries may help with magic numbers, yet the fuzzer has to figure out where to place such values. Researchers over the years have come up with a few approaches to handle magic numbers and checksums in grey-box fuzzers.

**Sub-instruction Profiling.** While understanding how a large amount of logic can be encoded in a single comparison is not trivial, one could break multi-byte comparisons into single-byte [1] (or even single-bit [22]) checks to better track progress when attempting to match constant values. LAF-INTEL [1] and COMPARECOVERAGE [20] are compiler extensions to produce binaries for such a fuzzing. HONGGFUZZ [28] implements this technique for fuzzing programs when the source is available, while AFL++ [16] can automatically transform binaries during fuzzing. STEELIX [21] resorts to static analysis to filter out likely uninteresting comparisons from profiling. While sub-instruction profiling is valuable to overcome tests for magic numbers, it is ineffective however with checksums.

**Taint Analysis.** Dynamic taint analysis (DTA) [26] tracks when and which input parts affect program instructions. VUZZER [25] uses DTA for checks on magic numbers in binary code, identified as comparison instructions where an operand is input-dependent

and the other is constant: VUZZER places the constant value in the input portion that propagates directly to the former operand. ANGORA [10] brings two improvements: it identifies magic numbers that are not contiguous in the input with a multi-byte form of DTA, and uses gradient descent to mutate tainted input bytes efficiently. It however requires compiler transformations to enable such mutations.

**Symbolic Execution.** DTA techniques can at best identify checksum tests, but not provide sufficient information to address them. Several proposals (e.g., [23, 27, 31, 35]) use symbolic execution [6] to identify and try to solve complex input constraints involved in generic roadblocks. TAINTSCOPE [31] identifies possible checksums with DTA, patches them away for the sake of fuzzing, and later tries to repair inputs with symbolic execution. It requires the user to provide specific seeds and is subject to false positives [23, 27].

T-Fuzz [23] makes a step further by removing the need for specific seeds and extends the set of disabled checks during fuzzing to any sanity (*non-critical*) check that is hard to bypass for a fuzzer but not essential for the computation. For instance, magic numbers can be safely ignored to help fuzzing, while a check on the length of a field should not be patched away. As in TAINTSCOPE, crashing inputs are repaired using symbolic execution or manual analysis.

DRILLER [27] instead uses symbolic execution as an alternate strategy to explore inputs, switching to it when fuzzing exhausts a budget obtaining no coverage improvement. Similarly, QSYM [35] builds on concolic execution implemented via dynamic binary instrumentation [13] to trade exhaustiveness for speed.

**Approximate Analyses.** A recent trend is to explore solutions that approximately extract the information that DTA or symbolic execution can bring, but faster.

REDQUEEN [5] builds on the observation that often input bytes flow directly, or after simple encodings (e.g. swaps for endianness), into instruction operands. This **input-to-state** (I2S) correspondence can be used instead of DTA and symbolic execution to deal with magic bytes, multi-byte compares, and checksums. REDQUEEN approximates DTA by changing input bytes with random values (*colorization*) to increase the entropy in the input and then looking for matching patterns between comparisons operands and input parts, which would suggest a dependency. When both operands of a comparison change, but only for one there is an I2S relationship, REDQUEEN deems it as a likely checksum test. It then patches the operation, and later attempts to repair the input with educated guesses consisting in mutations of the observed comparison operand values. A topological sort of patches addresses nested checksums.

SLF [34] attempts a dependency analysis to deal with the generation of valid input seeds when no meaningful test cases are available. It starts from a small random input and flips individual bits in each byte, executing the program to collect operand values involved in comparisons. When consecutive input bytes affect the same set of comparisons across the mutations, they are marked as part of the same field. SLF then uses heuristics on observed values to classify checks according to their relations with the exercised inputs: its focus are identifying offsets, counts, and length of input fields, as they can be difficult to reason about for symbolic executors.

In the context of concolic fuzzers, ECLIPSER [11] relaxes the path constraints over each input byte. It runs a program multiple times

**Table 1: Comparison with related approaches.**

Fuzzer	Binary	Magic bytes	Checksums	Chunk-based	Grammar-based	Automatic
AFL++	✓	✓	✗	✗	✗	✓
ANGORA	✗	✓	✗	✗	✗	✓
ECLIPSER	✓	✓	✗	✗	✗	✓
REDQUEEN	✓	✓	✓	✗	✗	✓
STEELIX	✓	✓	✗	✗	✗	✓
NAUTILUS	✗	✓	✗	✗	✓	✗
SUPERION	✓	✓	✗	✗	✓	✗
GRIMOIRE	✓	✓	✓	✗	✓	✓
AFLSMART	✓	✓	✗	✓	✗	✗
WEIZZ	✓	✓	✓	✓	✗	✓

by mutating individual input bytes to identify the affected branches. Then it collects constraints resulting from them, considering however only linear and monotone relationships, as other constraint types would likely require a full-fledged SMT solver. ECLIPSER then selects one of the branches and flips its constraints to generate a new input, mimicking dynamic symbolic execution [6].

### 2.3 Format-Aware Fuzzing

Classic CGF techniques lose part of their efficacy when dealing with structured input formats found in files. As mutations happen on bit-level representations of inputs, they can hardly bring the structural changes required to explore new compartments of the data processing logic of an application. Format awareness can however boost CGF: in the literature we can distinguish techniques targeting *grammar-based* formats, where inputs comply to a language grammar, and *chunk-based* ones, where inputs follow a tree hierarchy with C structure-like data chunks to form individual nodes.

**Grammar-Based Fuzzing.** LANGFUZZ [17] generates valid inputs for a Javascript interpreter using a grammar, combining in a black-box manner sample code fragments and test cases. NAUTILUS [4] and SUPERION [30] are recent grey-box fuzzer proposals that can test language interpreters without requiring a large corpus of valid inputs or fragments, but only an ANTLR grammar file.

GRIMOIRE [8] then removes the grammar specification requirement. Based on REDQUEEN, it identifies fragments from an initial set of inputs that trigger new coverage, and strips them from parts that would not cause a coverage loss. GRIMOIRE notes information for such gaps, and later attempts to recursively splice-in parts seen in other positions, thus mimicking grammar combinations.

**Chunk-Based Fuzzing.** SPIKE [3] lets users describe the network protocol in use for an application to improve black-box fuzzing. PEACH [15] generalizes this idea, applying format-aware mutations on an initial set of valid inputs using a user-defined input specification dubbed *peach pit*. As they are input-aware, some literature calls such black-box fuzzers *smart* [24]. However a smart grey-box variant may outperform them, as due to the lack of feedback (as in explored code) they do not keep mutating interesting inputs.

AFLSMART [24] validates this speculation by adding smart (or *high-order*) mutations to AFL that add, delete, and splice chunks in an input. Using a peach pit, AFLSMART maintains a *virtual structure* of the current input, represented conceptually by a tree whose internal nodes are chunks and leaves are attributes. Chunks are characterized by initial and end position in the input, a format-specific chunk type, and a list of children attributes and nested chunks. An attribute is a *field* that can be mutated without altering the structure. Chunk addition involves adding as sibling a chunk taken from another input, with both having a parent node of the

same type. Chunk deletion trims the corresponding input bytes. Chunk splicing replaces data in a chunk using a chunk of the same type from another input. As virtual structure construction is expensive, AFLSMART defers smart mutations based on the time elapsed since it last found a new path, as trying them over every input would make AFLSMART fall behind classic grey-box fuzzing.

## 2.4 Discussion

Tables 1 depicts where WEIZZ fits in the state of the art of CGF techniques mentioned in the previous sections. Modern general-purpose CGF fuzzers (for which we choose a representative subset with the first five entries) can handle magic bytes, but only REDQUEEN proposes an effective solution for generic checksums. In the context of grammar-based CGF fuzzers, GRIMOIRE is currently the only fully automatic (i.e., no format specification needed) solution, and can handle both magic bytes and checksums thanks to the underlying REDQUEEN infrastructure. With WEIZZ we bring similar enhancements to the context of chunk-based formats, proposing a scheme that at the same time can handle magic bytes and checksums and eliminates the need for a specification that characterizes AFLSMART.

## 3 METHODOLOGY

The fuzzing logic of WEIZZ comprises two stages, depicted in Figure 1. Both stages pick from a shared queue made of inputs processed by previous iterations of either stage. The surgical stage identifies dependencies between an input and the comparisons made in the program: it summarizes them by placing *tags* on input bytes, and applies deterministic mutations to the sole bytes that turn out to influence operands of comparison instructions. The structure-aware stage extends the nondeterministic working of AFL, leveraging previously assigned tags to infer the location of fields and chunks in an input and mutate them. In the next sections we will detail the inner workings of the two stages.

### 3.1 Surgical Stage

The surgical stage replaces the deterministic working of AFL, capturing in the process dependency information for comparison instructions that is within immediate reach of a fuzzer. WEIZZ summarizes it for the next stage by placing tags on input bytes, and uses it also to identify I2S relationships (Section 2.2) and checksums.

The analysis is context-sensitive, that is, when analyzing a comparison we take into account its calling context [14]: the *site* of the comparison is computed as the exclusive OR of the instruction address with the word used to encode the calling context.

WEIZZ also maintains a global structure *CI* to keep track of comparison instructions that the analysis of one or more inputs indicated as possibly involved in checksum tests.

We will use the running example of Figure 2 to complement the description of each component of the surgical stage. Before detailing them individually, we provide the reader with an overview of the workflow that characterizes this stage as an input enters it.

**3.1.1 Overview.** Given an input *I* picked from the queue, WEIZZ attempts to determine the dependencies between every bit of *I* taken individually and the comparison instructions in the program.

Procedure GETDEPS builds two data structures, both indexed by a hash function *Sites* for comparison sites. A comparison table

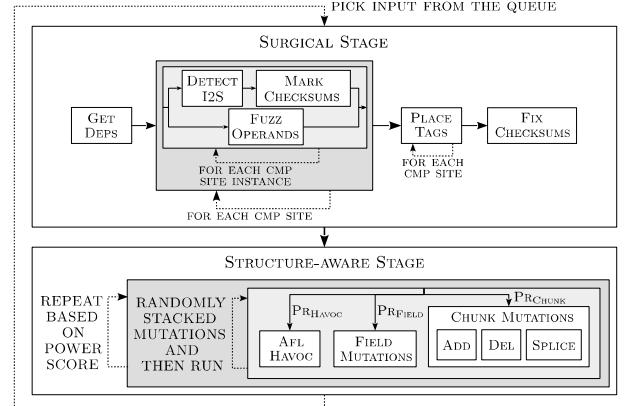


Figure 1: Two-stage architecture of WEIZZ.

*CT* stores values for operands involved in observed instances of comparison instructions at different sites. For such operands *Deps* stores which bytes in the input can influence them.

WEIZZ then moves to analyzing the recorded information. For each site, it iterates over each instance present in *CT* looking for I2S dependencies ( $\text{DETECTI2S} \mapsto R$ ) and checksum information ( $\text{MARKCHECKSUMS} \mapsto CI$ ), and mutates bytes that can alter comparison operands leveraging recorded values ( $\text{FUZZOPERANDS}$ ).

The stage then moves to tag construction, with each input byte initially untagged. It sorts comparison sites according to when they were first encountered and processes them in this order. Procedure PLACETAGS assigns a tag to each input byte taking into account I2S relationships *R*, checksum information *CI*, the initially computed dependencies *Deps*, and data associated with comparison sites *CT*.

Like other fuzzers [5, 31] WEIZZ forces checksums by patching involved instructions as it detects one, postponing to the end of the surgical stage the input repairing process FIXCHECKSUMS required to meet the unaltered checksum condition. To this end we use a technique very similar to the one of REDQUEEN (Section 2.2).

The output of the surgical stage is an input annotated with discovered tags and repaired to meet checksums, and enters the queue ready for the structure-aware stage to pick it up.

The procedures for which we report the name with underlined style are described only informally in this paper: the reader can find their full pseudocode in our extended online technical report<sup>1</sup>.

**3.1.2 Dependency Identification.** A crucial feature that makes a fuzzer effective is the ability to understand how mutations over the input affect the execution of a program. In this paper we go down the avenue of fast analyses to extract approximate dependency information. We propose a technique that captures which input bytes individually affect comparison instructions: it can capture I2S facts like the technique of REDQUEEN, but also non-I2S relationships that turn out to be equally important to assign tags to input bytes.

Algorithm 1 describes the GETDEPS procedure used to this end. Initially we run the program over the current input instrumenting all comparison instructions. The output is a comparison table *CT* that records the operands for the most recent  $|J|$  observations (*instances*) of a comparison site. For each comparison site *CT* keeps

<sup>1</sup><https://arxiv.org/abs/1911.00621>.

```

void parser(char* input, int len) {
    unsigned short id, size, expected, ck = 0;
    int i, offset = 0;
    id = read_ushort(input + offset);
    offset += sizeof(id);
    size = read_ushort(input + offset);
    offset += sizeof(size);
    CMP_A: if (id >= 0xAAAA) exit(1);
    CMP_B: if (size > len - 3*sizeof(short)) exit(1);
        offset += size;
    CMP_C: for (i = 0; i < offset; ++i)
        ck ^= input[i] << (i % 8);
        expected = read_ushort(input + offset);
    CMP_D: if (ck != expected) exit(1);

    // process id and data
}

```

(a)

(b) Input format: [id] [size] [data] [checksum]  
where: id, size, and checksum are 2-byte long  
data is size-byte long

Bytes of seed:  
[0E 00][02 00][41 41][36 0C]

Collected comparison instances:

CMP <sub>A</sub>	CMP <sub>B</sub>	CMP <sub>C</sub>	CMP <sub>D</sub>
E, AAAA	2, 2	0, 6	C36, C36
		1, 6	
		2, 6	
		3, 6	
		4, 6	
		5, 6	
		6, 6	

(c)

Bytes of seed after flipping first bit:  
[8E 00][02 00][41 41][36 0C]

Collected comparison instances:

CMP <sub>A</sub>	CMP <sub>B</sub>	CMP <sub>C</sub>	CMP <sub>D</sub>
<b>8E</b> , AAAA	2, 2	0, 6	<b>CB6</b> , C36
		1, 6	
		2, 6	
		3, 6	
		4, 6	
		5, 6	
		6, 6	

(d)

Figure 2: Example for surgical stage: (a) code of function parser; (b) input format; comparison instances collected by WEIZZ when running parser on (c) the initial seed and (d) on the seed after flipping its first bit (*affected operands are marked in bold*).

```

CT: Sites × J × {op1, op2} → V   [cmp site instance & operand → value]
Deps: Sites × J × {op1, op2} → A  [cmp site inst. & operand → array of |input| booleans]
function GETDEPS(I):
1 CT ← RUNINSTR(I)
2 foreach b ∈ {0...len(I)-1} do
3     Deps(s, j, op)[b] ← false ∀ (s, j, op) ∈ dom(CT)
4     foreach k ∈ {0...7} do
5         CT' ← RUNINSTR(BITFLIP(I, b, k))
6         foreach (s, j, op) ∈ dom(CT) do
7             if HITS(CT, s) ≠ HITS(CT', s) then continue
8             Deps(s, j, op)[b] ← Deps(s, j, op)[b] ∨ CT(s, j, op) ≠ CT'(s, j, op)
9 return CT, Deps

```

Algorithm 1: Dependency identification step.

a timestamp for when RUNINSTR first observed it, and how many times it encountered such site in the execution (HITS at line 7).

WEIZZ then attempts to determine which input bytes contribute to operands at comparison sites, either directly or through derived values. By altering bytes individually, WEIZZ looks for sites that see their operands changed after a mutation. The algorithm iterates over the bits of each byte, flipping them one at a time and obtaining a *CT'* for an instrumented execution with the new input (line 5). We mark the byte as a dependency for an operand of a comparison site instance (line 8) if its *b*-th byte has changed in *CT'* w.r.t. *CT*.

Bit flips may however cause execution divergences, as some comparisons likely steer the program towards different paths (and when such an input is *interesting* we add it to the queue, see Section 2.1). Before updating dependency information, we check whether a comparison site *s* witnessed a different number of instances in the two executions (line 7). This is a proxy to determine whether execution did not diverge locally at the comparison site. We prefer a local policy over enforcing that all sites in *CT* and *CT'* see the same number of instances. In fact for a mutated input some code portions can see their comparison operands change (suggesting a dependency) without incurring control-flow divergences, while elsewhere the two executions may differ too much to look for correlations.

**Example.** We discuss a simple parsing code (Figure 2a) for a custom input format characterized by three fields id, size, and checksum, each expressed using 2 bytes, and a field data of variable length encoded by the size field (Figure 2b).

Our parser takes as argument a pointer *input* to the incoming bytes and their number *len*. It assumes the input buffer will contain at least 6 bytes, which is the minimum size for a valid input holding empty data. The code operates as follows:

- it reads the id and size fields from the buffer;
- it checks the validity of field id (label CMP\_A);
- it checks whether the buffer contains at least 6+size bytes to host data and the other fields (label CMP\_B);
- it computes a checksum value iterating (label CMP\_C) over the bytes associated with id, size, and data;
- it reads the expected checksum from the buffer and validates the one presently computed (label CMP\_D).

Figure 2c shows the comparison instances collected by WEIZZ when running the function parser over the initial seed. For brevity we omit timestamps and hit counts and we assume that the calling context is not relevant, so we can use comparison labels to identify sites. Multiple instances appear for site CMP\_C as it is executed multiple times within a loop.

Figure 2d shows the comparison instances collected after flipping one bit in the first byte of the seed, with changes highlighted in bold. Through these changes WEIZZ detects that the first operand of both CMP\_A and CMP\_D was affected, revealing dependencies between these two comparison sites and the first input byte. Subsequent flips will lead to inputs that reveal further dependencies:

- the 1st operand of CMP\_A depends on the bytes of id;
- the 1st oper. of CMP\_B depends on the bytes of size;
- the 2nd oper. of CMP\_C depends on the 1st byte of size<sup>2</sup>;
- the 1st oper. of CMP\_D depends on the bytes of both id and data, as well as on the first byte of size, while the 2nd oper. is affected by the checksum field bytes;

3.1.3 Analysis of Comparison Site Instances. After constructing the dependencies, WEIZZ starts processing the data recorded for up to  $|J|$  instances of each comparison site. The first analysis is the detec-

<sup>2</sup>The dependency is exposed only when flipping the second least significant bit of size, turning it into the value zero, as flipping other bits makes the size invalid and aborts the execution prematurely at CMP\_B.

tion of I2S correspondences with DETECTI2S. REDQUEEN explores this concept to deal with roadblocks, based on the intuition that parts of the input can flow directly into the program state in memory or registers used for comparisons (Section 2.2). For instance, magic bytes found in headers are likely to take part in comparison instructions as-is or after some simple encoding transformation [5]. We apply DETECTI2S to every operand in a comparison site instance and populate the  $R$  data structure with newly discovered I2S facts.

MARKCHECKSUMS involves the detection of checksumming sequences. Similarly to REDQUEEN, we mark a comparison instruction as likely involved in a checksum if the following conditions hold: (1) one operand is I2S and its size is at least 2 bytes; (2) the other operand is not I2S and GETDEPS revealed dependencies on some input bytes; and (3)  $\bigwedge_b (\text{Deps}(s, j, \text{op}_1)[b], \text{Deps}(s, j, \text{op}_2)[b]) = \text{false}$ , that is, the sets of their byte dependencies are disjoint.

The intuition is that code compares the I2S operand (which we speculate to be the expected value) against a value derived from input bytes, and those bytes do not affect the I2S operand or we would have a circular dependency. We choose a 2-byte minimum size to reduce false positives. Like prior works we patch candidate checksum tests to be always met, and defer to a later stage both the identification of false positives and the input repairing required to meet the condition(s) from the original unpatched program.

Finally, FUZZOPERANDS replaces the deterministic mutations of AFL with surgical byte substitutions driven by data seen at comparison sites. For each input byte, we determine every CT entry (i.e., each operand of a comparison site instance) it influences. Then we replace the byte and, depending on the operand size, its surrounding ones using the value recorded for the other operand. The replacement can use such value as-is, rotate it for endianness, increment/decrement it by one, perform zero/sign extension, or apply ASCII encoding/decoding of digits. Each substitution yields an input added to the queue if its execution reveals a coverage improvement.

**Example.** When analyzing the function `parser`, WEIZZ is able to detect that the first operands of `CMP_A` and `CMP_B` and the second operand of `CMP_D` are I2S. On the other hand, the second operand of `CMP_C` and the first operand of `CMP_D` are not I2S, although they depend on some of the input bytes.

WEIZZ marks `CMP_D` as a comparison instance likely involved into a checksum since the three required conditions hold: the second operand is I2S and has size 2, the first operand is not I2S but depends on several input bytes, and the two operands show dependencies on disjoint sets of bytes.

FUZZOPERANDS attempts surgical substitutions based on observed operand values: for instance, it can place as-is value `0xAAAA` recorded at the comparison site `CMP_A` in the bytes for the field `id` and trigger the enclosed `exit(1)` statement.

**3.1.4 TAG PLACEMENT.** WEIZZ succinctly describes dependency information between input bytes and performed comparisons by annotating such bytes with tags essential for the subsequent structural information inference. For the  $b$ -th input byte  $\text{Tags}[b]$  keeps:

- $id$ : (the address of) the comparison instruction chosen as most significant among those  $b$  influences;
- $ts$ : the timestamp of the  $id$  instruction when first met;

- $parent$ : the comparison instruction that led WEIZZ to the most recent tag assignment prior to  $\text{Tags}[b]$ ;
- $dependsOn$ : when  $b$  contains a checksum value, the comparison instruction for the innermost nested checksum that verifies the integrity of  $b$ , if any;
- $flags$ : stores the operand affected by  $b$ , if it is I2S, or if it is part of a checksum field;
- $numDeps$ : the number of input bytes that the operand in  $flags$  depends on.

The tag assignment process relies on spatial and temporal locality in how comparison instructions get executed. WEIZZ tries to infer structural properties of the input based on the intuition that a program typically uses distinct instructions to process structurally distinct items. We can thus tag input bytes as related when they reach one same comparison directly or through derived data. When more candidates are found, we use temporal information to prioritize instructions with a lower timestamp, as input format validation normally takes place in parsing code from the early stages of the execution. As we explain next, we extend this scheme to account for checksums and to reassign tags when heuristics spot a likely better candidate than the current choice. Temporal information can serve also as proxy for hierarchical relationships with parent tags.

Algorithm PLACETAGS iterates over comparison sites sorted by when first met in the execution, and attempts an inference for each input byte. We apply it to individual  $op$  instruction operands seen at a site  $s$ . If for the current byte  $b$  we find no dependency among all recorded instances  $\text{Deps}(s, j, op)$ , the cycle advances to the next byte, otherwise we compute a  $numDeps$  candidate, i.e., the number  $n$  of input bytes that affect the instruction, computed as  $n \leftarrow \sum_k (1 \text{ if } \bigvee_j \text{Deps}(s, j, op)[k] \text{ else } 0)$  where  $k$  indexes the input length. If the byte is untagged we tag it with the current instruction, otherwise we consider a reassignment. If the current tag  $\text{Tags}[b]$  does not come from a checksum test and  $n$  is smaller than  $\text{Tags}[b].numDeps$ , we reassign the tag as fewer dependencies suggest the instruction may be more representative for the byte.

When we find a comparison treating the byte as from a checksum value, we always assign it as its tag. To populate the  $dependsOn$  field we use a topological sort of the dependencies  $\text{Deps}$  over each input byte, that is, we know when a byte part of a checksum value represents also a byte that some outer checksum(s) verify for integrity. We later repair inputs starting from the innermost checksum, with  $dependsOn$  pointing to the next comparison to process.

**Example.** Let us consider the seed of Figure 2c. The analysis of dependencies leads to the following tag assignments:

- bytes from field `id` affect sites `CMP_A` and `CMP_D`: `CMP_A` is chosen as tag as it is met first in the execution;
- bytes from `size` affect `CMP_B`, `CMP_C`, and `CMP_D`: `CMP_B` is chosen as tag as it temporally precedes the other sites;
- bytes from `data` and `checksum` affect only `CMP_D`, which becomes the `id` for their respective tags: however the tags will differ in the `flags` field as those for checksum bytes are marked as involved in a checksum field.

**3.1.5 Checksum Validation & Input Repair.** The last surgical step involves checksum validation and input repair for checksums patched

		id	start-1	start	start+1	end-1	end	end+1
		ts	1	5	5	5	5	16
Pattern ① field vs one multi-byte cmp	int p = &input[x]; if (p == magic)							
	char * p = &input[x]; if (p[0] == m[3] && p[1] == m[2] && p[2] == m[1] && p[3] == m[0])		B	A	E	G	B	D
Pattern ② field vs one cmp per byte	short * p = &input[x]; if (p[0] == -1 && p[1] == 0xABCD)	Tags	1	5	6	7	8	16
Pattern ③ field vs multi-byte cmps		id	B	A	A	G	G	D
		ts	1	5	5	6	6	16

Figure 3: Field patterns identified by WEIZZ.

in the program during previous iterations. As the technique is conceptually similar to the one of REDQUEEN, we discuss it briefly.

Algorithm `FIXCHECKSUMS` uses topologically sorted tags that were previously marked as involved in checksum fields. For each tag, it first extracts the checksum computed by the program (i.e., the input-derived operand value of the comparison) and where in the input the contents of the other operand (which is necessarily I2S, see Section 3.1.3) are stored, then it replaces such bytes with the expected value. It then disables the involved patch and runs the program on the repaired input: if it handled the checksum correctly, we will observe the same execution path as before, otherwise the checksum was a false positive and we have to bail.

At the end of the process WEIZZ re-applies patches that were not a false positive: this will benefit both the second stage and future surgical iterations over other inputs. It then also applies patches for checksums newly discovered by `MARKCHECKSUMS` in the current surgical stage, so when it will analyze again the input (or similar ones from `FUZZOPERANDS`) it will be able to overcome them.

### 3.2 Structure-Aware Stage

The second stage of WEIZZ generates new inputs via nondeterministic mutations that can build on tags assigned in the surgical stage. Like in the AFL realm, the number of derived inputs depends on an energy score that the power scheduler of AFL assigns to the original input [24]. Each input is the result of a variable number (1 to 256) of stacked mutations, with WEIZZ picking nondeterministically at each step a *havoc*, a *field*, or a *chunk* mutation scheme. The choice of keeping the havoc mutations of AFL is shared with previous chunk-oriented works [24], as combining them with structure-aware mutations improves the scalability of the approach.

Field mutations build on tags assigned in the surgical stage to selectively alter bytes that together likely represent a single piece of information. Chunk mutations target instead higher-level, larger structural transformations driven by tags.

As our field and chunk inference strategy is not sound, especially compared to manually written specifications, we give WEIZZ leeway in the identification process. WEIZZ never reconstructs the input structure in full, but only identifies individual fields or chunks in a nondeterministic manner when performing a mutation. The downside of this choice is that WEIZZ may repeat work or make conflicting choices among mutations, as it does not keep track of past choices. On the bright side, WEIZZ does not commit to possibly erroneous choices when applying one of its heuristics: this limits their impact to the subsequent mutations, and grants more leeway to the overall fuzzing process in exploring alternate scenarios.

```

function FIELDMUTATION(Tags, I):
1   b  $\leftarrow$  pick u.a.r. from {0 ... len(I)}
2   for i  $\in$  {b ... len(I) - 1} do
3     if Tags[i].id == 0 then continue
4     start  $\leftarrow$  GoLEFTWHILESAMETAG(Tags, i)
5     with probab. PT2S or if !I2S(Tags[start]) then
6       end  $\leftarrow$  FINDFIELDEND(Tags, start, 0)
7       I  $\leftarrow$  MUTATE(I, start, end); break
8   return I

```

Algorithm 2: Field identification and mutation.

We will describe next how we use input tags to identify fields and chunks and the mutations we apply. At the end of the process, we execute the program over the input generated from the stacked mutations, looking for crashes and coverage improvements. We promptly repair inputs that cause a new crash, while the others undergo repair when they eventually enter the surgical stage.

**3.2.1 Fields.** Field identification is a heuristic process designed around prominent patterns that WEIZZ should identify. The first one is straightforward and sees a program checking a field using a single comparison instruction. We believe it to be the most common in real-world software. The second one instead sees a program comparing every byte in a field using a different instruction, as in the following fragment from the `lodepng` library:

```

unsigned char lodepng_chunk_type_equals(const unsigned char*
    chunk, const char* type) {
  if (strlen(type) != 4) return 0;
  return (chunk[4]==type[0] && chunk[5]==type[1] && chunk[6]==
    type[2] && chunk[7]==type[3]); }

```

which checks each byte in the input string *chunk* using a different comparison statement. Such code for instance often appears in program to account for differences in endianness. The two patterns may be combined to form a third one, as in the bottom part of Figure 3, that we consider in our technique as well.

Let us present how WEIZZ captures the patterns instantiated in Figure 3. For the first pattern, since a single instruction checks all the bytes in the field, we expect to see the corresponding input bytes marked with the same tag. For the second pattern, we expect instead to have consecutive bytes marked with different tags but consecutive associated timestamps, as no other comparison instruction intervenes. In the figure we can see a field made of bytes with tag ids {A, E, G, B} having respective timestamps {5, 6, 7, 8}. The third pattern implies having (two or more) subsequences made internally of the same tag, with the tag changing across them but with a timestamp difference of one unit only.

Procedure `FIELDMUTATION` (Algorithm 2) looks nondeterministically for a field by choosing a random position *b* in the input. If there is no tag for the current byte, the cursor advances until it reaches a tagged byte (line 3). On the contrary, if the byte is tagged WEIZZ checks whether it is the first byte in the candidate field or there are any preceding bytes with the same tag, retracting to the leftmost in the latter case (line 4). With the start of the field now found, WEIZZ evaluates whether to mutate it: if the initial byte is I2S, the mutation happens only with a probability as such a byte could represent a magic number. Altering a magic number would lead to program paths handling invalid inputs, which in general are less appealing for a fuzzer. The extent of the mutation is decided by the helper procedure `FINDFIELDEND`, which looks for sequences of

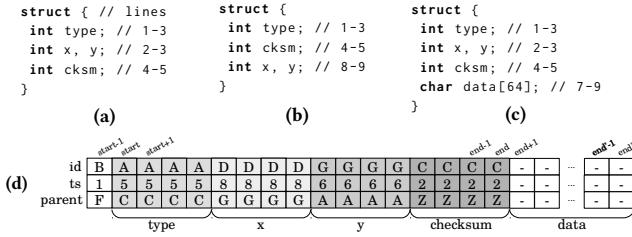


Figure 4: Examples of chunks that WEIZZ can look for.

tagged bytes that meet one of the three patterns discussed above. FIELDMUTATION then alters the field by choosing one among the twelve AFL length-preserving havoc transformations over its bytes.

**3.2.2 CHUNKS.** WEIZZ heuristically locates plausible boundaries for chunks in the input sequence, then it applies higher-order mutations to candidate chunks. As running example we discuss variants of a structure representative of many chunks found in binary formats.

The first variant (Figure 4a) has four fields: a type assigned with some well-known constant, two data fields  $x$  and  $y$ , and a  $\text{cksm}$  value for integrity checking. Let us consider a program that first computes the checksum of the structure and compares it against the namesake field, then it verifies the type and  $x$  fields in this order, executes possibly unrelated comparisons (i.e., they do not alter tags for the structure bytes), and later on makes a comparison depending on  $y$ . For the sake of simplicity we assume that field processing matches the first pattern seen in the previous section (i.e., one comparison per field). The output of PLACETAGS will be coherent with the graphical representation of Figure 4d.

We now describe how our technique is able to identify the boundaries of this chunk using tags and their timestamps. Similarly as with fields, we pick a random position  $b$  within the input and analyze the preceding bytes, retracting the cursor as long as the tag stays the same. For any  $b$  initially falling within the 4 bytes of type, WEIZZ finds the same *start* index for the chunk. To find the end position it resorts to FINDCHUNKEND (Algorithm 3).

The procedure initially recognizes the 4 bytes of type (line 1), then recursively looks for adjacent fields when their timestamps are higher than the one for the current field (lines 2-3). This recursive step reveals fields  $x$  and  $y$ . The  $\text{cksm}$  field has a lower timestamp value (as the program checked it before analyzing type), but lines 4-5 can include it in the chunk as they inspect parent data. The parent is the comparison from the tag assignment prior to the current one, and the first activation of FINDCHUNKEND will see that the tag of the first byte of  $\text{cksm}$  matches the parent for the tag for type.

To explain lines 6-9 in Algorithm 3, let us consider variants of the structure that exercise them. In Figure 4b  $\text{cksm}$  comes before type, and in this case the algorithm would skip over lines 2-3 without incrementing  $end$ , thus missing  $x$  and  $y$ . Lines 8-9 can add to the chunk bytes from adjacent fields as long as they have increasing timestamps with respect to the one from the tag for the  $k$ -th byte (the first byte in type in this case). In Figure 4c we added an array data of 64 bytes to the structure: it may happen that WEIZZ leaves binary blobs untagged if the program does not make comparisons over them or some derived values. Line 7 can add such sequences to a chunk, extending  $end$  to the new  $end'$  value depicted in Figure 4d.

```

function FINDCHUNKEND(Tags, k):
1   end ← GoRIGHTWHILESAMETAG(Tags, k)
2   while Tags[end+1].ts >= Tags[k].ts do
3     end ← FINDCHUNKEND(Tags, end+1)
4   while Tags[end+1].id == Tags[k].parent do
5     end ← end + 1
6   with probability Pextend do
7     while Tags[end+1].id == 0 do end ← end + 1
8     while Tags[end+1].ts >= Tags[k].ts do
9       end ← FINDCHUNKEND(Tags, end+1)
10  return end

```

Algorithm 3: Boundary identification for chunks.

With FINDCHUNKEND we propose an inference scheme inspired by the layout of popular formats. The first field in a chunk is typically also the one that the program analyzes first, and we leverage this fact to speculate where a chunk may start. If the program verifies a checksum before accessing even that field, we link it to the chunk using parent information, otherwise “later” checksums represent a normal data field. Lines 7-9 enlarge chunks to account for different layouts and tag orderings, but only with a probability to avoid excessive extension. The algorithm can also capture partially tagged blobs through the recursive steps it can take.

Observe however that in some formats there might be dominant chunk types, and choosing the initial position randomly would reveal a chunk of non-dominant type with a smaller probability. We devise an alternate heuristic that handles this scenario better: it randomly picks one of the comparison instructions appearing in tags, assembles non-overlapping chunks with FINDCHUNKEND starting at a byte tagged by such an instruction, and picks one among the built chunks. As WEIZZ is unaware of the format characteristics, we choose between the two heuristics with a probability.

For the current chunk selection, WEIZZ attempts one of the following higher-order mutations (Section 2.3):

- **Addition.** It adds a chunk from another input to the chunk that encloses the current one. WEIZZ picks a tagged input  $I'$  from the queue and looks for chunks in it starting with bytes tagged with the same parent field of the leading bytes in the current chunk. It picks one randomly and extends the current input by adding its associated bytes before or after the current chunk. The parent tag acts as proxy for the nesting information available instead to AFLSMART.
- **Deletion.** It removes the input bytes for the chunk.
- **Splicing.** It picks a similar chunk from another input to replace the current one. It scans that input looking for chunks starting with the same tag (AFLSMART uses type information) of the current, randomly picks one, and substitutes the bytes.

**3.2.3 Discussion.** We can now elaborate on why, in order to back the field and chunk mutations just described, we cannot rely on the dependency identification techniques of REDQUEEN or SLF.

Let us assume REDQUEEN colors an input with a number of  $A$  bytes and initially logs a  $\text{cmp } A, B$  operation. REDQUEEN would attempt to replace each occurrence of  $A$  with  $B$  and validate it by looking for coverage improvements when running the program over the new input, this time with logging disabled. This strategy works well if the goal are only I2S replacements, but for field identification there are two problems: (i) with multiple  $B$  bytes in the input, we cannot determine which one makes a field for the comparison,

and (ii) if  $B$  is already in another input in the queue, no coverage improvement comes from the replacement, and REDQUEEN misses a direct dependency for the current input. Our structural mutations require tracking comparisons all along for I2S and non-I2S facts.

SLF can identify dependencies similarly to our GETDEPS, but recognizes only specialized input portions—that we can consider fields—based on how the program treats them. SLF supports three categories of program checks and can mutate portions e.g. by replicating those involved in a count check (§2.2). While WEIZZ may do that through chunk addition, it also mutates fields—and whole chunks—that are not treated by SLF.

## 4 IMPLEMENTATION

We implemented our ideas on top of AFL 2.52b and QEMU 3.1.0 for x86-64 Linux targets. For branch coverage and comparison tables we use a shadow call stack to compute context-sensitive information [14], which may let a fuzzer explore programs more pervasively [10]. We index the coverage map using the source and destination basic block addresses and a hash of the call stack.

Natural candidates for populating comparison tables are `cmp` and `sub` instructions. We store up to  $|J|=256$  entries per site. Like REDQUEEN we also record `call` instructions that may involve comparator functions: we check whether the locations for the first two arguments contain valid pointers, and dump 32 bytes for each operand. Treating such calls as operands (think of functions that behave like `memcmp`) may improve the coverage, especially when the fuzzer is configured not to instrument external libraries.

An important optimization involves *deferring* surgical fuzzing with a timeout-based mechanism, letting inputs jump to the second stage with a decreasing probability if an interesting input was discovered in the current window (sized as 50 seconds).

Note that untagged inputs can only undergo havoc mutations in the second stage: we thus introduce *derived* tags, which are an educated guess on the actual tag based on tags seen in a similar input. Derived tags speed up our fuzzer, and actual tags replace them when the input eventually enters the surgical stage.

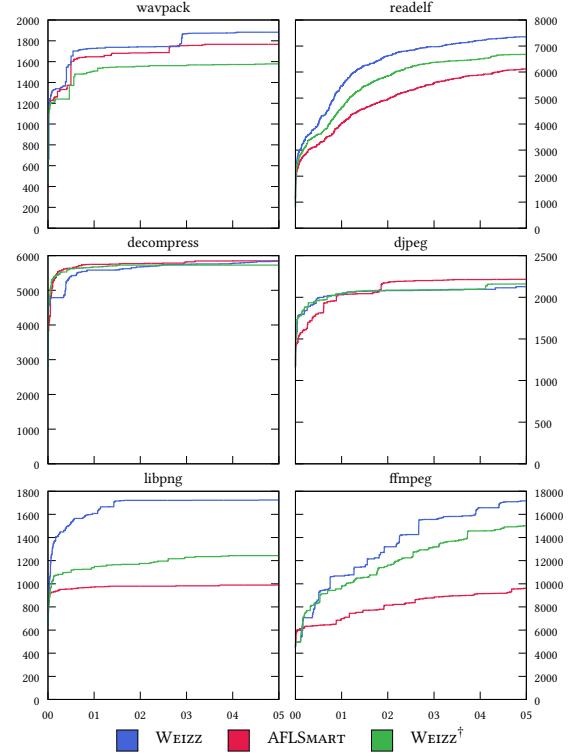
Two scenarios can give rise to derived tags. One case happens when starting from an input  $I$  the surgical mutations of FuzzOPERANDS( $I$ ) produce one or more inputs  $I'$  that improve coverage and are added to the queue. Once PlaceTags has tagged  $I$ , we copy such tags to  $I'$  (structurally analogous to  $I$ , as FuzzOPERANDS operates with “local” mutations) and mark them as derived.

Similarly, a tagged input  $I_1$  can undergo high-order mutations that borrow bytes from a tagged input  $I_2$ , namely addition or splicing. In this case the added/spliced bytes of the mutated  $I_1'$  coming from  $I_2$  get the same tags seen in  $I_2$ , while the remaining bytes keep the tags seen for them in  $I_1$ .

## 5 EVALUATION

In our experiments we tackle the following research questions:

- **RQ 1.** How does WEIZZ compare to state-of-the-art fuzzers on targets that process chunk-based formats?
- **RQ 2.** Can WEIZZ identify new bugs?
- **RQ 3.** How do tags relate to the actual input formats? And what is the role of structural mutations and roadblock bypassing in the observed improvements?



**Figure 5: Basic block coverage over time (5 hours).**

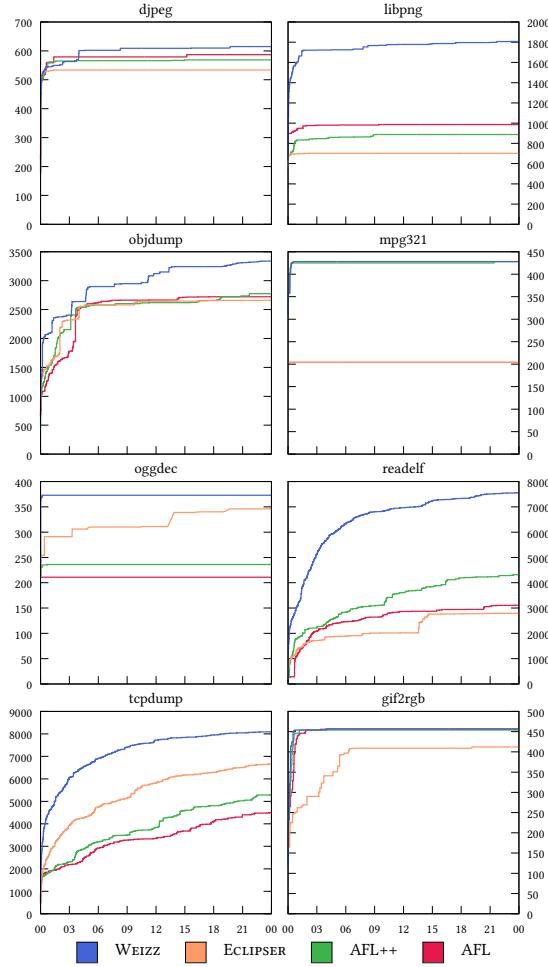
**Benchmarks.** We consider the following programs (version, input format): wavpack (5.1.0, WAV), decompress (2.3.1, JP2), ffmpeg (4.2.1, AVI), libpng (1.6.37, PNG), readelf (2.3.0, ELF), djpeg (5db6a68, JPEG), objdump (2.31.51, ELF), mpg321 (MP3, 0.3.2), oggdec (1.4.0, OGG), tcpcdump (4.9.2, PCAP), and gif2rgb (5.2.1, GIF). The first 6 programs were considered in past evaluation of AFLSMART, with a format specification available for it. The last 8 programs are commonly used in evaluations of general-purpose fuzzers.

**Experimental setup.** We ran our tests on a server with two Intel Xeon E5-4610v2@2.30GHz CPUs and 256 GB of RAM, running Debian 9.2. We measured the cumulative basic block coverage of different fuzzers from running an application over the entire set of inputs generated by each fuzzer. We repeated each experiment 5 times, plotting the median value in the charts. For the second stage of WEIZZ we set  $\text{Pr}_{field} = \text{Pr}_{chunk} = 1/15$  (Figure 1), similarly to the probability of applying smart mutations in AFLSMART in [24].

### 5.1 RQ1: Chunk-Based Formats

We compare WEIZZ against the state-of-the-art solution for chunk-based formats AFLSMART, which applies higher-order mutations over a virtual input structure (Section 2.3). We then take into account general-purpose fuzzers that previous studies [11, 24] suggest as being still quite effective in practice on this type of programs.

**5.1.1 AFLSMART.** For AFLSMART we used its release 604c40c and the peach pits written by its authors for the virtual input structures involved. We measured the code coverage achieved by: (a) AFLSMART with stacked mutations but without deterministic stages as suggested in its documentation, (b) WEIZZ in its standard config-



**Figure 6: Basic block coverage over time (24 hours).**

uration, and (c) a variant  $\text{WEIZZ}^\dagger$  with FuzzOPERANDS, checksum patching, and input repairing disabled.  $\text{WEIZZ}^\dagger$  lets us focus on the effects of tag-based mutations, giving up on roadblock bypassing capabilities missing in AFLSMART. We provide (a) and (c) with a dictionary of tokens for format under analysis like in past evaluations of AFLSMART. We use AFL test cases as seeds, except for wavpack and ffmpeg where we use minimal syntactically valid files [9].

Figure 5 plots the median basic block coverage after 5 hours. Compared to AFLSMART,  $\text{WEIZZ}$  brings appreciably higher coverage on 3 out of 6 subjects (readelf, libpng, ffmpeg), slightly higher on wavpack, comparable on decompress, and slightly worse on djpeg. To understand these results, we first consider where  $\text{WEIZZ}^\dagger$  lies, and then discuss the benefits from the additional features of  $\text{WEIZZ}$ .

Higher coverage in  $\text{WEIZZ}^\dagger$  comes from the different approach to structural mutations. AFLSMART follows a format specification, while we rely on how a program handles the input bytes:  $\text{WEIZZ}^\dagger$  can reveal behaviors that characterize the actual implementation and that may not be necessarily anticipated by the specification. The first implication is that  $\text{WEIZZ}^\dagger$  mutates only portions that the program has already processed in the execution, as tags derive from executed comparisons. The second is that imprecision in in-

ferring structural facts may actually benefit  $\text{WEIZZ}^\dagger$ . The authors of AFLSMART acknowledge how relaxed specifications can expose imprecise implementations [24]: we will return to this in Section 6.

$\text{WEIZZ}^\dagger$  is a better alternative than AFLSMART for readelf, libpng, and ffmpeg. When we consider the techniques disabled in  $\text{WEIZZ}^\dagger$ ,  $\text{WEIZZ}$  brings higher coverage for two reasons. I2S facts help  $\text{WEIZZ}$  replace magic bytes only in the right spots compared to dictionaries, which can also be incomplete. This turns out important also in multi-format applications like ffmpeg. Then, checksum bypassing allows it to generate valid inputs for programs that check data integrity, such as libpng that computes CRC-32 values over data.

We also consider the collected crashes, found only for wavpack and ffmpeg. In the first case, the three fuzzers found the same bugs. For ffmpeg,  $\text{WEIZZ}$  found a bug from a division by zero in a code section handling multiple formats: we reported the bug and its developers promptly fixed it. The I2S-related features of  $\text{WEIZZ}$  were very effective for generating inputs that deviate significantly from the initial seed, e.g., mutating an AVI file into an MPEG-4 one. In Section 5.2 we back this claim with one case study.

**5.1.2 Grey-Box Fuzzers.** We now compare  $\text{WEIZZ}$  against 8 popular applications handling chunk-based formats, heavily tested by the community [2], and used in past evaluations of state-of-the-art fuzzers [5, 21, 24, 25]. We tested the following fuzzers: (a) AFL 2.53b in QEMU mode, (b) AFL++ 2.54c in QEMU mode enabling the CompareCoverage feature, and (c) ECLIPSER with its latest release when we ran our tests (commit 8a00591) and its default configuration. As  $\text{WEIZZ}$  targets binary programs, we rule out fuzzers like ANGORA that require source instrumentation. For sub-instruction profiling, since STEELIX is not publicly available, we choose AFL++ instead.

While considering REDQUEEN could be tantalizing, its hardware-assisted instrumentation could make an unfair edge as here we compare coverage of QEMU-based proposals, as the different overheads may mask why a given methodology reaches some coverage. We attempt an experiment in Section 5.3 by configuring  $\text{WEIZZ}$  to resemble its features, and compare the two techniques in Section 6.

As the competitors we consider have no provisions for structural mutations, we opted for a larger budget of 24 hours in order to see whether they could recover in terms of coverage over time. Consistently with previous evaluations [11], we use as initial seed a string made of the ASCII character "0" repeated 72 times. However, for libpng and tcpdump we fall back to a valid initial input test (not\_kitty.png from AFL for libpng and a PCAP of a few seconds for tcpdump) as no fuzzer, excluding  $\text{WEIZZ}$  on libpng, achieved significant coverage with artificial seeds. We also provide AFL with format-specific dictionaries to aid it with magic numbers.

Figure 6 plots the median basic block coverage over time.  $\text{WEIZZ}$  on 5 out of 8 targets (libpng, oggdec, tcpdump, objdump and readelf) achieves significantly higher code coverage than other fuzzers. The first three process formats with checksum fields that only  $\text{WEIZZ}$  repairs, although ECLIPSER appears to catch up over time for oggdec. Structural mutation capabilities combined with this factor may explain the gap between  $\text{WEIZZ}$  and other fuzzers. For objdump and readelf, we may speculate I2S facts are boosting the work of  $\text{WEIZZ}$  by the way REDQUEEN outperforms other fuzzers over them in its evaluation [5] (in objdump logging function arguments was crucial [5]). On mpg321 and gif2rgb, AFL-based fuzzers perform

very similarly, with a large margin over ECLIPSER, confirming that standard AFL mutations can be effective on some chunk-based formats. Finally, WEIZZ leads for jpeg but the other fuzzers are not too far from it. Overall, AFL is interestingly the best alternative to WEIZZ for jpeg, libpng, and gif2rgb. When taking into account crashes, WEIZZ and AFL++ generated the same crashing inputs for mpg321, while only WEIZZ revealed a crash for objdump.

## 5.2 RQ2: New Bugs

To explore its effectiveness, we ran WEIZZ for 36 hours on several real-world targets, including the processing of inputs not strictly adhering to the chunk-based paradigm. WEIZZ revealed 16 bugs in 9 popular, well-tested applications: objdump, CUPS (2 bugs), libmimage (2), dimg2img (3), jbig2enc, mpg321, ffmpeg (3 in libavformat, 1 in libavcodec), sleuthkit, and libvmdk. Overall 6 bugs are NULL pointer dereferences (CWE-476), 1 involves an unaligned realloc (CWE-761), 2 can lead to buffer overflows (CWE-122), 2 cause an out-of-bounds read (CWE-125), 2 a division by zero (CWE-369), and 3 an integer overflow (CWE-190). We detail two interesting ones.

**CUPS.** While the HTML interface of the Common UNIX Printing System is not chunk-oriented, we explored if WEIZZ could mutate the HTTP requests enclosing it. WEIZZ crafted a request that led CUPS to reallocate a user-controlled buffer, paving the road to a *House of Spirit* attack [7]. The key to finding the bug was to have FUZZOPERANDS replace some current input bytes with I2S facts ('Accept-Language' logged as operand in a call to `_cups_strcasecmp`), thus materializing a valid request field that chunk mutations later duplicated. Apple confirmed and fixed the bug in CUPS v2.3b8.

**libMirage.** We found a critical bug in a library providing uniform access to CD-ROM image formats. An attacker can generate a heap-based buffer overflow that in turn may corrupt allocator metadata, and even grant root access as the CDEmu daemon using it runs with root privileges on most Linux systems. We used an ISO image as initial seed. WEIZZ exposed the bug in the NRG image parser, demonstrating how it can deviate even considerably among input formats based exclusively on how the code handles input bytes.

## 5.3 RQ3: Understanding the Impact of Tags

Programs can differ significantly in how they process input bytes of different formats, yet we find it interesting to explore *why* WEIZZ can be effective on a given target. We discuss two case studies where we seek how tags can assist field and chunk identification in libpng, and we see how smart mutations and roadblocks bypassing are essential for ffmpeg, but either alone is not enough for efficacy.

**Tag Accuracy.** Figure 7 shows the raw bytes of the seed `not_kitty.png` for libpng. It starts with a 8-byte magic number (in orange) followed by 5 chunks, each characterized by four fields: length (4 bytes, in green), type (4 bytes, red), data (heterogeneous, as with `PNG_CHUNK_IHDR` data in yellow), and a CRC-32 checksum (4 bytes, in blue). The last chunk has no data as it marks the file end.

To understand the field identification process, we analyzed the tags from the surgical stage for the test case. Starting from the first byte, we apply FINDFIELDEND repeatedly at every position after the end of the last found field. In Figure 7 we delimit each found field in square brackets, and underline bytes that WEIZZ deems from

[89 50 4E 47 0D 0A 1A 0A]	[00] [00 00 OD]	[49 48 44 52]
[00 00 00 20 00 00 00 00]	[20 08 03 00 00 00 00 00]	[44 A4 8A]
[C6] [00] [00 00 19]	[74 45 58 74]	[53 6F 66 74 77 61 72]
[65] [00] [41 64 6F 62 65 20 49]	[6D 61 67 65 52 65 61]	
64 79] [71 C9 65 3C]	[00] [00 00 0F]	[60 40 54 45] [66 CC]
CC] [FF FF FF 00 00 00]	[33 99 66 99 FF CC]	[3E 4C AF]
[15] [00] [00 00 61]	[49 44 41 54]	
20 0C 03 93 98 FF BF B9 34 14 09 D4 3A 61 61 AO	78 DA DC 93 31 OE CO	
37 B0 F8 24 0B 0B [44] 13 44 D5 02 6E C1 0A 47 CC		
05 A1] EO [A7] [82 6F 17 08 CF] [BA] [54 A8 21 30 1A 6F		
01] [F0 93] [56 B4 3C 10] 7A 4D 20 4C] [F9 B7] [30 E4 44		
48 96 44 22 4C] [43 EE A9 38 F6 C9 D5 9B 51 6C E5		
F3 26 5C] [02 OC 00] [D2 66 03 35] [B0 D7 CB 9A] 00 00		
00 00	[49 45 4E 44 AE 42 60 82]	

□: field delimiters from WEIZZ    HEX: checksum field from WEIZZ  
 PNG fields: `PNG_SIGNATURE` `LENGTH` `TYPE` `PNG_CHUNK_IHDR` `CRC-32`  
 IDAT DATA `PNG_CHUNK_TEXT_LABEL` `PNG_CHUNK_TEXT_DATA` `PNG_CHUNK_PLTE`

Figure 7: Fields identified in the `not_kitty.png` test case.

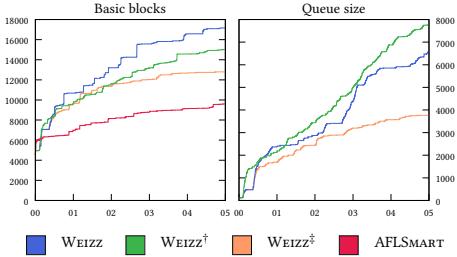


Figure 8: Analysis of ffmpeg with three variants of WEIZZ.

checksum fields. WEIZZ identifies correct boundaries with a few exceptions. The last three fields in the file are not revealed, as libpng never accesses that chunk. In some cases a data field is merged with the adjacent checksum value. Initially libpng does not make comparisons over the data bytes, but first computes their checksum and verifies it against the expected value with a comparison that will characterize also the data bytes (for the data-flow dependency). Dependency analysis on the operands (Section 3.1.3) however identifies the checksum correctly for FIXCHECKSUMS, and once the input gets repaired and enters the surgical stage again, libpng will execute new comparisons over the data bytes, and WEIZZ can identify the data field correctly tagging them with such instructions. Finally, the grey parts (IDAT DATA) represent a binary blob fragmented into distinct fields by WEIZZ with gaps from untagged bytes, as not all the routines that process them make comparisons over the blob.

Analyzing chunk boundaries is more challenging, as FINDFIELDEND is sensitive to the initial position. For instance, when starting the analysis at the 8-byte magic number it correctly identifies the entire test case as one chunk, or by starting at a length field it may capture all the other fields for that chunk<sup>3</sup>. However, when starting at a type field it may build an incomplete chunk. Nonetheless, even under imprecise boundary detection WEIZZ can still make valuable mutations for two reasons. First, mutations that borrow bytes from other test cases check the same leading tag, and this yields for instance splicing of likely “compatible” incomplete chunks. Second, this may be beneficial to exercise not so well-tested corner cases or shared parsing code: we will return to it in Section 6. For libpng we achieved better coverage than other fuzzers, including AFLSMART.

<sup>3</sup>Depending on the starting byte, it could miss some of the first bytes.

**Structural Mutations vs. Roadblocks.** To dig deeper in the experiments of Section 5.1, one question could be: *how much coverage improvement is due to structural mutations or roadblock bypassing?* We consider ffmpeg as case study: in addition to WEIZZ and WEIZZ<sup>†</sup> that lacks roadblock bypassing techniques but leverages structural mutations, we introduce a variant WEIZZ<sup>‡</sup> that can use I2S facts to bypass roadblocks like REDQUEEN but lacks tag-based mutations. In Figure 8 we report code coverage and input queue size after a run of 5 hours<sup>4</sup>. When taken individually, tag-based mutations seem to have an edge over roadblock bypass techniques (+17% coverage). This may seem surprising as ffmpeg supports multiple formats and I2S facts can speed up the identification of their magic numbers, unlocking several program paths. Structural mutations however affect code coverage more appreciably on ffmpeg. When combining both features in WEIZZ, we get the best result with 34% more coverage than in WEIZZ<sup>‡</sup>. Interestingly, WEIZZ<sup>†</sup> shows a larger queue size than WEIZZ: this means that although WEIZZ explores more code portions, WEIZZ<sup>†</sup> covers some of the branches more exhaustively, i.e., it is able to cover more hit count buckets for some branches.

## 6 CONCLUDING REMARKS

WEIZZ introduces novel ideas for computing byte dependency information to simultaneously overcome roadblocks and back fully automatic fuzzing of chunk-based binary formats. The experimental results seem promising: we are competitive with human-assisted proposals, and we found new bugs in well-tested software.

Our approach has two practical advantages: fuzzers already attempt bitflips in deterministic stages, and instrumenting comparisons is becoming a common practice for roadblocks. We empower such analyses to better characterize the program behavior while fuzzing, enabling the tag assignment mechanism. Prior proposals do not offer sufficient information to this end: even for REDQUEEN, its colorization [5] identifies I2S portions of an input (crucial for roadblocks) but cannot reveal dependencies for non-I2S bytes.

A downside is that bit flipping can get costly over large inputs. However, equally important is the time the program takes to execute one test case. In our experiments WEIZZ applied the surgical stage to inputs up to 3K bytes, with comparable or better coverage than the other fuzzers we considered. We leave to future work using forms of bit-level DTA [32, 33] as an alternative for “costly” inputs.

WEIZZ may miss dependencies for comparisons made with uninstrumented instructions. This can happen in optimized code that uses arithmetic and logical operations to set CPU flags for a branch decision. We may resort to intra-procedural static analysis to spot them [25] (as logging all of them blindly can be expensive) but currently opted for tolerating some inconsistencies in the heuristics we use, for instance skipping over one byte in FINDCHUNKEND when the remaining bytes would match the expected patterns.

Our structure-aware stage, in addition to not requiring a specification, is different than AFLSMART also in where we apply high-order operators. AFLSMART mutates chunks in a black-box fashion, that is, it has no evidence whether the program manipulated the involved input portion during the execution. WEIZZ chooses among chunks that the executed comparison instructions indirectly reveal. We find hard to argue which strategy is superior in the general case.

<sup>4</sup> AFLSMART shown as reference—queues are incomparable (no context sensitivity).

Another important difference is that, as our inference schemes are not sound, we may mutate inputs in odd ways, for instance replacing only portions of a chunk with a comparable counterpart from another input. In the AFLSMART paper the authors explain that they could find some bugs only thanks to a relaxed specification [24]. We find this consistent with the GRIMOIRE experience with grammars, where paths outside the specification revealed coding mistakes [8].

As future work we plan to consider a larger pool of subjects, and shed more light on the impact of structure-aware techniques: how they impact coverage, which are the most effective for a program, and how often the mutated inputs do not meet the specification. Answering these questions seems far from trivial due to the high throughput and entropy of fuzzing. There is also room to extend chunk inference with new heuristics or make the current ones more efficient. For instance, we are exploring with promising results a variant where we locate the beginning of a chunk at a field made of I2S bytes, possibly indicative of a magic value for its type.

## REFERENCES

- [1] 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. <https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/>. [Online; accessed 10-Sep-2019].
- [2] 2019. Google OSS-Fuzz: continuous fuzzing of open source software. <https://github.com/google/oss-fuzz>. [Online; accessed 10-Sep-2019].
- [3] Dave Aitel. 2002. The Advantages of Block-Based Protocol Analysis for Security Testing. [https://www.immunitysec.com/downloads/advantages\\_of\\_block\\_based\\_analysis.html](https://www.immunitysec.com/downloads/advantages_of_block_based_analysis.html). [Online; accessed 10-Sep-2019].
- [4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS*. <https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/>
- [5] Cornelius Aschermann, Sergei Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*. <https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/>
- [6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018), 39 pages. <https://doi.org/10.1145/3182657>
- [7] Blackngel. 2019. MALLOC DES-MALEFICARUM. <http://phrack.org/issues/66/10.html>. [Online; accessed 10-Sep-2019].
- [8] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002. <https://www.usenix.org/system/files/sec19-blazytko.pdf>
- [9] Mathias Bynens. 2019. Smallest possible syntactically valid files of different types. <https://github.com/mathiasbynens/small>. [Online; accessed 10-Sep-2019].
- [10] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. <https://doi.org/10.1109/SP.2018.00046>
- [11] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Greybox Concolic Testing on Binary Code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 736–747. <https://doi.org/10.1109/ICSE.2019.00082>
- [12] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. 391–402. <https://doi.org/10.1145/1455770.1455820>
- [13] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. 15–27. <https://doi.org/10.1145/3321705.3329819>
- [14] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. Mining Hot Calling Contexts in Small Space. *Software: Practice and Experience* 46 (2016), 1131–1152. <https://doi.org/10.1002/spe.2348>
- [15] M. Eddington. [n.d.]. Peach fuzzing platform. <https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html>. [Online; accessed 10-Sep-2019].

- [16] Marc Heuse, Heiko Eißfeldt, and Andrea Fioraldi. 2019. AFL++. <https://github.com/vanhauer-thc/AFLplusplus>. [Online; accessed 10-Sep-2019].
- [17] Christian Holler, Kim Herzog, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (SEC'12)*. 38–38. <http://dl.acm.org/citation.cfm?id=2362793.2362831>
- [18] Matthias Höschle and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 720–725. <https://doi.org/10.1145/2970276.2970321>
- [19] Matthias Höschle and Andreas Zeller. 2017. Mining Input Grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. 31–34. <https://doi.org/10.1109/ICSE-C.2017.14>
- [20] Mateusz Jurczyk. 2019. CompareCoverage. <https://github.com/googleprojectzero/CompareCoverage/>. [Online; accessed 10-Sep-2019].
- [21] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 627–637. <https://doi.org/10.1145/3106237.3106295>
- [22] Tavis Ormandy. 2009. Making Software Dumber. [http://taviso.decsystem.org/making\\_software\\_dumber.pdf](http://taviso.decsystem.org/making_software_dumber.pdf). [Online; accessed 10-Sep-2019].
- [23] H. Peng, Y. Shoshitaishvili, and M. Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. <https://doi.org/10.1109/SP.2018.00056>
- [24] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). <https://doi.org/10.1109/TSE.2019.2941681>
- [25] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/>
- [26] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*. 317–331. <https://doi.org/10.1109/SP.2010.26>
- [27] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23th Annual Network and Distributed System Security Symposium, NDSS*. <https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf>
- [28] Robert Swiecki. 2017. honggfuzz. <https://github.com/google/honggfuzz>. [Online; accessed 10-Sep-2019].
- [29] Ari Takanen, Jared D. Demott, and Charles Miller. 2018. *Fuzzing for Software Security Testing and Quality Assurance* (2nd ed.). Artech House, Inc., Norwood, MA, USA.
- [30] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 724–735. <https://doi.org/10.1109/ICSE.2019.00081>
- [31] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy (SP)*. 497–512. <https://doi.org/10.1109/SP.2010.37>
- [32] B. Yadegari and S. Debray. 2014. Bit-Level Taint Analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 255–264. <https://doi.org/10.1109/SCAM.2014.43>
- [33] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy (SP)*. 674–691. <https://doi.org/10.1109/SP.2015.47>
- [34] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 712–723. <https://doi.org/10.1109/ICSE.2019.00080>
- [35] Insoo Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. 745–761. <http://dl.acm.org/citation.cfm?id=3277203.3277260>
- [36] Michał Zalewski. 2019. American Fuzzy Lop. <https://github.com/Google/AFL>. [Online; accessed 10-Sep-2019].
- [37] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. 2019. The Fuzzing Book. <https://www.fuzzingbook.org/>. [Online; accessed 10-Sep-2019].

# Active Fuzzing for Testing and Securing Cyber-Physical Systems

Yuqi Chen

Singapore Management University  
Singapore  
yuqichen@smu.edu.sg

Bohan Xuan

Zhejiang University  
China  
xuanbohan@zju.edu.cn

Christopher M. Poskitt

Singapore Management University  
Singapore  
cposkitt@smu.edu.sg

Jun Sun

Singapore Management University  
Singapore  
junsun@smu.edu.sg

Fan Zhang\*

Zhejiang University  
China  
fanzhang@zju.edu.cn

## ABSTRACT

Cyber-physical systems (CPSs) in critical infrastructure face a pervasive threat from attackers, motivating research into a variety of countermeasures for securing them. Assessing the effectiveness of these countermeasures is challenging, however, as realistic benchmarks of attacks are difficult to manually construct, blindly testing is ineffective due to the enormous search spaces and resource requirements, and intelligent fuzzing approaches require impractical amounts of data and network access. In this work, we propose *active fuzzing*, an automatic approach for finding test suites of packet-level CPS network attacks, targeting scenarios in which attackers can observe sensors and manipulate packets, but have no existing knowledge about the payload encodings. Our approach learns regression models for predicting sensor values that will result from sampled network packets, and uses these predictions to guide a search for payload manipulations (i.e. bit flips) most likely to drive the CPS into an unsafe state. Key to our solution is the use of *online active learning*, which iteratively updates the models by sampling payloads that are estimated to maximally improve them. We evaluate the efficacy of active fuzzing by implementing it for a water purification plant testbed, finding it can automatically discover a test suite of flow, pressure, and over/underflow attacks, all with substantially less time, data, and network access than the most comparable approach. Finally, we demonstrate that our prediction models can also be utilised as countermeasures themselves, implementing them as anomaly detectors and early warning systems.

## CCS CONCEPTS

- Computer systems organization → Embedded and cyber-physical systems;
- Security and privacy → Intrusion detection systems;
- Computing methodologies → Active learning settings;

\*Also with Zhejiang Lab, and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397376>

## KEYWORDS

Cyber-physical systems; fuzzing; active learning; benchmark generation; testing defence mechanisms

## ACM Reference Format:

Yuqi Chen, Bohan Xuan, Christopher M. Poskitt, Jun Sun, and Fan Zhang. 2020. Active Fuzzing for Testing and Securing Cyber-Physical Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397376>

## 1 INTRODUCTION

Cyber-physical systems (CPSs), characterised by their tight and complex integration of computational and physical processes, are often used in the automation of critical public infrastructure [78]. Given the potential impact of cyber-attacks on these systems [46, 51, 60], ensuring their security and protection has become a more important goal than ever before. The different temporal scales, modalities, and process interactions in CPSs, however, pose a significant challenge for solutions to overcome, and have led to a variety of research into different possible countermeasures, including ones based on anomaly detection [11, 15, 27, 45, 48, 52, 58, 61, 66, 69, 71], fingerprinting [12, 13, 44, 56], invariant-based monitoring [7, 8, 10, 18, 24, 25, 28, 39, 82], and trusted execution environments [74].

Assessing how effective these different countermeasures are at detecting and preventing attacks is another challenge in itself. A typical solution is to use established benchmarks of attacks [2, 41], which have the advantage of facilitating direct comparisons between approaches, e.g. as done so in [52, 58, 61]. Such benchmarks, unfortunately, are few and far between: constructing them manually requires a great deal of time and expertise in the targeted CPS (all while risking insider bias), and generalising them from one CPS to another is a non-starter given the distinct processes, behaviours, and complexities that different systems exhibit.

An alternative solution is to generate benchmarks using *automated testing and fuzzing*, with these techniques overcoming the complexity of CPSs by having access to machine learning (ML) models trained on their data (e.g. logs of sensor readings, actuator, states, or network traffic). Existing solutions of this kind, however, tend to make unrealistic assumptions about an attacker's capabilities, or require a large body of training data that might not be available. The fuzzer of [26], for example, can automatically identify actuator configurations that drive the physical states of CPSs to different extremes, but the technology assumes the attacker to

have total control of the network and actuators, and is underpinned by a prediction model trained on complete sets of data logs from *several days* of operation. Blindly fuzzing without such a model, however, is ineffective at finding attacks: first, because the search spaces of typical CPSs are *enormous*; and second, because of the wasted time and resources required to be able to observe the effects on a system’s physical processes.

In this paper, we present *active fuzzing*, an automatic approach for finding test suites of packet-level CPS network attacks, targeting scenarios in which training data is limited, and in which attackers can observe sensors and manipulate network packets but have no existing knowledge about the encodings of their payloads. Our approach constructs regression models for predicting future sensor readings from network packets, and uses these models to guide a search for payload manipulations that systematically drive the system into unsafe states. To overcome the search space and resource costs, our solution utilises (*online*) *active learning* [63], a form of supervised ML that iteratively re-trains a model on examples that are estimated to maximally improve it. We apply it to CPSs by flipping bits of existing payloads in a way that is guided by one of two frameworks: Expected Model Change Maximization [17], and a novel adaptation of it based on maximising behaviour change. We query the effects of sampled payloads by spoofing them in the network, updating the model based on the observed effect.

We evaluate our approach by implementing it for the Secure Water Treatment (SWaT) testbed [4], a scaled-down version of a real-world water purification plant, able to produce up to five gallons of drinking water per minute. SWaT is a complex multi-stage system involving chemical processes such as ultrafiltration, de-chlorination, and reverse osmosis. Communication in the testbed is organised into a layered network hierarchy, in which we target the ring networks at the ‘lowest’ level that exchange data using EtherNet/IP over UDP. Our implementation manipulates the binary string payloads of 16 different types of packets, which when considered together have up to  $2^{2752}$  different combinations.

Despite the enormous search space, we find that active fuzzing is effective at discovering packet-level flow, pressure, and over/underflow attacks, achieving comparable coverage to an established benchmark [41] and an LSTM-based fuzzer [26] but with substantially less training time, data, and network access. Furthermore, by manipulating the bits of payloads directly, active fuzzing bypasses the logic checks enforced by the system’s controllers. These attacks are more sophisticated than those of the LSTM-based fuzzer [26], which can only generate high-level actuator commands and is unable to manipulate such packets. Finally, we investigate the utility of the learnt models in a different role: defending a CPS directly. We use them to implement anomaly detection and early warning systems for SWaT, finding that when models are suitably expressive, they are effective at detecting both random and known attacks.

**Summary of Contributions.** We present active fuzzing, a black-box approach for automatically discovering packet-level network attacks on real-world CPSs. By iteratively constructing a model with active learning, we demonstrate how to overcome enormous search spaces and resource costs by sampling new examples that maximally improve the model, and propose a new algorithm that guides this process by seeking maximally different behaviour. We evaluate the



**Figure 1: The Secure Water Treatment (SWaT) testbed**

efficacy of the approach by implementing it for a complex real-world critical infrastructure testbed, and show that it achieves comparable coverage to an established benchmark and LSTM-based fuzzer but with significantly less data, time, and network access. Finally, we show that the learnt models are also effective as anomaly detectors and early warning systems.

**Organisation.** In Section 2, we introduce the SWaT testbed, with a particular focus on its network and the structure of its packets. In Section 3, we present the components of our active fuzzing approach, and explain how to implement it both in general and for SWaT. In Section 4, we evaluate the efficacy of our approach at finding packet-level attacks, and investigate secondary applications of our models as anomaly detectors and early warning systems. In Section 5, we discuss some related work, then conclude in Section 6.

## 2 BACKGROUND

In the following, we present an overview of SWaT, a water treatment testbed that forms the critical infrastructure case study we evaluate active fuzzing on. We describe in more detail its network hierarchy and the structure of its packets, before stating the assumptions we make about the capabilities of attackers.

**SWaT Testbed.** The CPS forming the case study of this paper is Secure Water Treatment (SWaT) [4], a scaled-down version of a real-world water purification plant, able to produce up to five gallons of safe drinking water per minute. SWaT (Figure 1) is intended to be a *testbed* for advancing cyber-security research on critical infrastructure, with the potential for successful technologies to be transferred to the actual plants it is based on. The testbed has been the subject of multiple hackathons [9] involving researchers from both academia and industry, and over the years has established a benchmark of attacks to evaluate defence mechanisms against [41].

SWaT treats water across multiple distinct but co-operating stages, involving a variety of complex chemical processes, such as de-chlorination, reverse osmosis, and ultrafiltration. Each stage in the CPS is controlled by a dedicated Allen-Bradley ControlLogix programmable logic controller (PLC), which communicates with the sensors and actuators relevant to that stage over a ring network, and with other PLCs over a star network. Each PLC cycles through

```

###[ Ethernet ]##
dst      = e4:90:69:a3:0c:f6
src      = 00:1d:9c:c8:03:e7
type     = IPv4
###[ IP ]##
version  = 4
ihl      = 5
tos      = 0xbc
len      = 68
id       = 18067
flags    =
frag     = 0
ttl      = 64
proto    = udp
chksum   = 0xb1f3
src      = 192.168.0.10
dst      = 192.168.0.12
\options \
###[ UDP ]##
sport    = 2222
dport    = 2222
len      = 48
checksum = 0xfcfa2
###[ ENIP_CPF ]##
count    = 2
\items \
|###[ CPF_AddressDataItem ]##
| type_id  = Sequenced Address Item
| length   = 8
|###[ CPF_SequencedAddressItem ]##
| connection_id= 469820023
| sequence_number= 18743
|###[ CPF_AddressDataItem ]##
| type_id  = Connected Transport Packet
| length   = 22
|###[ Raw ]##
| load     = '-2\x01\x00\x00\x00\x00\x00\x00
                        \r\x00\x00\x00\x00\x00\x00\x00
                        \x00\x00\x00\x00\x00\x00\x00
                        \x00'

```

**Figure 2: A SWaT packet after dissection by Scapy**

its program, computing the appropriate commands to send to actuators based on the latest sensor readings received as input. The system consists of 42 sensors and actuators in total, with sensors monitoring physical properties such as tank levels, flow, pressure, and pH values, and actuators including motorised valves (for opening an inflow pipe) and pumps (for emptying a tank). A historian regularly records the sensor readings and actuator commands during SWaT's operation. SCADA software and tools developed by Rockwell Automation are available to support some analyses.

The sensors in SWaT are associated with manufacturer-defined ranges of *safe* values, which in normal operation, they are expected to remain within. If a sensor reports a (true) reading outside of this range, we say the physical state of the CPS has become *unsafe*. If a level indicator transmitter, for example, reports that the tank in stage one has become more than a certain percentage full (or empty), then the physical state has become unsafe due to the risk of an overflow (or underflow). Unsafe pressure states indicate the risk of a pipe bursting, and unsafe levels of water flow indicate the risk of possible cascading effects in other parts of the system.

SWaT implements a number of standard safety and security measures for water treatment plants, such as alarms (reported to the operator) for when these thresholds are crossed, and logic checks for commands that are exchanged between the PLCs. In addition, several attack defence mechanisms developed by researchers have been installed (see Section 5).

The network of the SWaT testbed is organised into a layered hierarchy compliant with the ISA99 standard [53], providing different

levels of segmentation and traffic control. The ‘upper’ layers of the hierarchy, Levels 3 and 2, respectively handle operation management (e.g. the historian) and supervisory control (e.g. touch panel, engineering workstation). Level 1 is a star network connecting the PLCs, and implements the Common Industrial Protocol (CIP) over EtherNet/IP. Finally, the ‘lowest’ layer of the hierarchy is Level 0, which consists of ring networks (EtherNet/IP over UDP) that connect individual PLCs to their relevant sensors and actuators.

Tools such as Wireshark [6] and Scapy [3] can be used to dissect the header information of a Level 0 SWaT packet, as illustrated in Figure 2. Here, the source IP address (192.168.0.10) and target IP address (192.168.0.12) correspond respectively to PLC1 and its remote IO device. Actuator commands (e.g. “open valve MV101”) are encoded in the binary string payloads of these packets. In Figure 2, the payload is 22 bytes long, but Level 0 packets can also have a payload length of 10 or 32 bytes. Randomly manipulating the payloads has limited use given the size of the search space ( $2^{2752}$  possibilities when considering the 16 types of packets we sample; see Section 3.1). Our solution uses active learning to overcome this enormous search space, establishing how different bits impact the physical state without requiring any knowledge of the encoding.

**Attacker Model.** In this work, we assume that attackers have knowledge of the network protocol (e.g. EtherNet/IP over UDP at Level 0 of SWaT), and thus are able to intercept (unencrypted) packets, dissect their header information, and manipulate their payloads. We assume that the packet payloads are binary strings, but *do not* assume any existing knowledge about their meaning or encoding schemes. We assume that attackers can always access the ‘true’ sensor readings while the system is operating, in order to be able to observe the effects of a packet manipulation, or to judge whether or not an attack was successful. These live sensor readings can be observed over several minutes at a time in order to perform some pre-training and active learning, but in contrast to other approaches (e.g. [26]), we do not require access to extensive sets of data for offline learning, and we do not require the ability to arbitrarily issue high-level actuator commands across the system—we do so *only* by manipulating payloads.

### 3 ACTIVE FUZZING

Our approach for automatically finding packet-level network attacks in CPSs consists of the following steps. First, data is *collected*: packets are sniffed from the network, their payloads are extracted, and (true) sensor readings are queried. Second, we *pre-train* initial regression models, that take concatenations of packet payloads and predict the future effects on given sensors. Third, we apply an *online active learning* framework, iteratively improving the current model by sampling payloads estimated to maximally improve it. Finally, we search for candidate attacks by flipping important bits in packet payloads, and using our learnt models to identify which of them will drive the system to a targeted unsafe state.

Algorithm 1 presents the high-level algorithm of these steps for active fuzzing. Note that the notation in Line 8 indicates concatenation of sequences. In particular,  $t_s$  copies of the vector  $p$  are appended to sequence  $P$  to add additional weight to the new example when the model is re-trained.

**Algorithm 1:** High-Level Overview of Active Fuzzing

---

**Input:** Sensor  $s$ , prediction time  $t_s$ , pre-training time  $t_p$   
**Output:** Prediction model  $M_s$

- 1 Sniff packets and observe values of  $s$  for  $t_p$  minutes;
- 2 Construct a sequence  $P$  of feature (bit-)vectors from packet payloads;
- 3 Construct a sequence  $V$  such that each  $V[i]$  contains the value of  $s$  observed  $t_s$  seconds after  $P[i]$  was sniffed;
- 4 (Pre-)train a regression model  $M_s$  predicting  $V$  from  $P$ ;
- 5 **repeat**
- 6     Sample a new feature vector  $p$  using an active learning framework (Section 3.2);
- 7     Wait for  $t_s$  seconds then observe the value  $v_s$  of  $s$ ;
- 8      $P := P \cup \langle p \rangle^{t_s}$ ; [concatenation of  $t_s$  copies]
- 9      $V := V \cup \langle v_s \rangle^{t_s}$ ;
- 10 **until** timeout;
- 11 Re-train  $M_s$  to predict  $V$  from  $P$ ;
- 12 **return** model  $M_s$ ;

---

In the following, we describe the steps of the algorithm in more detail, and present the details of one particular implementation for the SWaT water purification testbed.

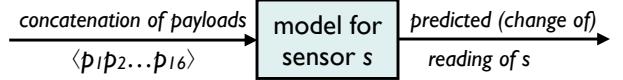
### 3.1 Packet Sniffing and Pre-Training

**Collecting Raw Data.** Both the pre-training and active learning phases of our approach require access to two types of data from the CPS under test. First, they must be able to sniff the network packets and extract their binary string payloads. Second, they must be able to access the true readings of any sensors under consideration, as the idea is to be able to observe the effects on sensor readings of different payload manipulations.

For SWaT, our approach extracts packets from Level 0 of the network hierarchy, i.e. the packets exchanged between PLCs and remote IO devices. By targeting this lowest level of the network, we ensure that our manipulations are altering the actuator states directly. For our prototype, we physically connect some Raspberry Pis to the PLCs of SWaT and sniff the packets using a network connection bridge; in reality, an attacker might sniff packets by other means, e.g. exploiting the wireless connection when enabled. As Level 0 implements the EtherNet/IP protocol, we can use the tcpdump packet analyser to capture packets, and Scapy to further extract their contents.

For our prototype, we obtain the current sensor readings by querying SWaT's historian. We assume that the historian's data is true, i.e. that the system is not simultaneously under attack by another entity, and that it is operationally healthy. In reality, an attacker might access this information through an exploit in the historian, e.g. an EternalBlue exploit [1], or a backdoor connection (both of which were discovered in SWaT hackathons [9]).

**Pre-Training Models.** A goal of our approach is to minimise the amount of data required to train an accurate prediction model for sensor readings. We thus proceed in two phases: a pre-training phase, and an active learning phase. The pre-training phase uses network data to construct an *initial* prediction model, the idea being that it provides a reasonable enough starting point such that



**Figure 3: Input/output of a learnt model for sensor  $s$**

active learning will later converge. A key distinction between the two stages is how the attacker behaves: while pre-training, they sit silently to observe *normal* packets of the system; but while actively learning, they intervene by injecting (possibly) *abnormal* packets and then observe the effects. It is thus important to minimise the amount of the time spent in the latter phase to avoid detection.

We require a series of regression models, one per sensor, that take as input the payloads of captured packets, and return as output a prediction of how the considered (true) sensor reading will evolve after a time period. To achieve this goal requires a number of system-specific decisions to be made, for example, the types of packets to train the model on, and a fixed time period that is appropriate to the processes involved (some will change the physical state more quickly than others). There are several types of regression models that are fit for the task. In this work, we focus on two: *linear models* and *gradient-boosting decision trees (GBDT)* [35]. A linear model is the simplest possible choice and thus serves as our baseline, whereas the GBDT is a well-known and popular example of a non-linear model. Both models can be integrated with existing active learning frameworks for regression, which was a key reason for their selection. Several more expressive models, such as neural networks, do not have any good online active learning strategies (to the best of our knowledge).

In SWaT, packets are collected from Level 0 (see Section 2) in the first four stages of the system. By observing the network traffic, we identified four different types of packets in each stage based on payload lengths and headers: packets that have payloads of length (1) 10 bytes; (2) 32 bytes; (3) 22 bytes, with a source IP address of 192.168.0.S0; and (4) 22 bytes, with a source IP address of 192.168.0.S2. Here,  $S$  is replaced with the given stage of the system (1, 2, 3, or 4). Across these four stages, there are thus 16 different types of packets in total. In constructing a feature vector for training, we make no assumptions about the meaning of these different packets, so select the first of *each* type of packet that is collected at a particular time point and concatenate their payloads together in a fixed order. This leads to feature vectors containing a series of 2752 bits.

Along with constructing a sufficient number of feature vectors (experimentally determined in Section 4), we also query the historian for sensor values after fixed time periods have passed. For flow and pressure sensors this time period is 5 seconds; for tank level sensors, it is 30 seconds, owing to the fact that they change state rather more slowly. With this data collected, we train linear and GBDT models for each individual sensor in turn, such that a sensor reading can be predicted for a given bit vector of payloads from the 16 types. An overview of the input/output of these models is given in Figure 3. Note that for flow and pressure sensors, the corresponding models predict their future *values*, whereas for tank level sensors, the corresponding models predict by how much they

will change. This discrepancy is due to the fact that the effects of flow/pressure attacks stabilise at a final value very quickly.

### 3.2 Active Learning and Attack Discovery

**Active Learning.** After completing the pre-training phase, we should now have a model that is capable of making some reasonable predictions with respect to *normal* packets in the CPS network. However, the attacks we need for testing the CPS are not necessarily composed of normal packets. We need to train the model further on a broader set of examples, but cannot do it blindly owing to the expense of running the system and the enormity of the search space ( $2^{2752}$  potential combinations of feature vectors in SWaT).

Our solution is to train the model further using (*online*) *active learning* [63], a supervised ML approach that iteratively improves the current model. Theoretical studies have shown that active learning may exponentially reduce the amount of training data needed, e.g. [33, 37, 38]. The idea is to reduce the amount of additional data by sampling examples that are *estimated* to maximally change the current model in some way. In our case, we use one of two active learning frameworks to guide the construction of new feature vectors by flipping the bits of existing ones (this is more conservative than constructing payloads from scratch, but minimises the possibility of packet rejection). Once new feature vectors have been sampled, we can decompose them into their constituent packets, spoof them in the network, observe their effects on true sensor readings, then re-train the model accordingly.

While active learning for classification problems is well-studied, there are limited active learning frameworks for regression, and some of the ones that exist make assumptions unsuitable for our application (e.g. a Gaussian distribution [21]). However, the Expected Model Change Maximization (EMCM) approach of Cai et al. [17] avoids this assumption and is suitable for CPSs. Their framework is based on the idea of sampling new examples that are estimated to maximally change the model itself, i.e. the gradient in linear models, or a linear approximation of GBDTs based on ‘super features’ extracted from the trees (see [17] for details).

Inspired by EMCM, and motivated by the fact we can query live behaviour of the system, we also propose a variant of the framework called Expected Behaviour Change Maximisation (EBCM). Instead of sampling examples estimated to maximally change the model, EBCM attempts to identify examples that cause maximally different *behaviour* from what the system is currently exhibiting. For example, if a considered sensor reading is increasing, then EBCM may identify examples that cause it to decrease as much as possible instead. The intuition of the approach is that exploring different behaviour in a particular context is more informative. It also seeks to check that unfamiliar packets predicted to cause that behaviour *really do* cause that behaviour, updating the model otherwise.

Algorithm 2 summarises the steps of EBCM, in which a new feature vector is constructed by sampling additional packets, randomly flipping the bits of several copies, and choosing a vector that would have led to a maximally different reading than the original. Note that to ensure some variation, the feature vector is chosen from a set of several using Roulette Wheel Selection [43], which assigns to each candidate a probability of being selected based on its ‘fitness’, here defined as the absolute difference between what the sensor

---

**Algorithm 2:** Expected Behaviour Change Maximisation

---

**Input:** Prediction model  $M_s$ , prediction time  $t_s$ , maximum number of bits to flip  $n_m$   
**Output:** Feature (bit-)vector  $p_f$

- 1 Sniff current packets and construct a feature vector  $p_o$  based on their payloads;
- 2 Wait for  $t_s$  seconds then observe the value  $v_s$  of  $s$ ;
- 3 Let  $P := \langle \rangle$ ; [empty sequence]
- 4 Let  $D := \langle \rangle$ ;
- 5 **repeat**
- 6     Construct a new vector  $p$  from  $p_o$  by randomly selecting and flipping  $n \leq n_m$  bits;
- 7      $v_p := M_s(p)$ ;
- 8      $P := P \cup \langle p \rangle$ ;
- 9      $D := D \cup \langle |v_s - v_p| \rangle$ ;
- 10 **until** timeout;
- 11 Select a feature vector  $p_f$  from  $P$  using *Roulette Wheel Selection* with corresponding fitness values in  $D$ ;
- 12 **return** feature (bit-)vector  $p_f$ ;

---

reading actually became (with respect to the original packets) and what the current model predicted for the candidate. If  $f_i$  is the fitness of one of  $n$  candidates, then its probability of being selected is  $f_i / \sum_{j=1}^n f_j$ . A random number is generated between 0 and the sum of the candidates’ fitness scores. We then iterate through the candidates until the accumulated fitness is larger than that number, returning that final candidate as our chosen bit-vector.

For SWaT, we implemented both EMCM and EBCM, using the same construction of feature vectors (i.e. a concatenation of the payloads of 16 types of packets). Upon computing new feature vectors using these active learning frameworks, we then break the vectors down into their constituent packets, and spoof them in Level 0 of the network using Scapy. After spoofing, we wait either 5 or 30 seconds (when targeting flow/pressure or tank level sensors respectively) before querying the latest sensor value, then re-train the model based on the new packets and readings observed. This process is repeated until a suitable timeout condition (Section 4.2).

**Attack Discovery and Fuzzing.** In the final step of our approach, we use the learnt models (Figure 3) to discover attacks, i.e. packet manipulations that will drive targeted (true) sensor readings out of their safe operational ranges. In particular, after choosing a sensor to target, the corresponding model is used to evaluate a number of candidate packet manipulations and reveal the one that is (predicted) to realise the attack most effectively. The final part of our approach consists of *generating* those candidate packet manipulations for the model to evaluate.

Algorithm 3 presents the steps of our packet manipulation procedure for attack discovery. The idea of the algorithm is to identify the bits that are most *important* (i.e. have the most influence in the prediction), generate candidates by flipping fixed numbers of those bits, before broadening the search to other, less important bits too. As different candidates are generated, they are evaluated against a simple objective function that is maximised as the predicted sensor state becomes closer to an edge of its safe operational range. Suppose that  $v_s$  denotes a value of sensor  $s$ , and that  $L_s$  and  $H_s$

**Algorithm 3:** Attack Discovery

---

**Input:** Prediction model  $M_s$ , number of bits to flip  $n$ , objective function  $f$

**Output:** A bit-vector  $p_{max}$

- 1 Sniff current packets and construct a feature vector  $p_o$  based on their payloads;
- 2 Construct a sequence  $\Phi$  of (0-based) indices of  $p_o$ , from the position with the highest *feature importance* (Section 3.2) to the lowest;
- 3  $k := n - 1$ ;
- 4  $Done := \emptyset$ ;
- 5  $f_{max} := 0$ ;
- 6 **repeat**
- 7      $\Phi_k := \{i \mid i \in \Phi[0..k]\}$ ;
- 8      $Combs := \{B \mid B \in 2^{\Phi_k} \wedge |B| = n \wedge B \notin Done\}$ ;
- 9     **for**  $c \in Combs$  **do**
- 10         Construct  $p$  from  $p_o$  by flipping  $p_o[i]$  for every  $i \in c$ ;
- 11         **if**  $f(M_s(p)) > f_{max}$  **then**
- 12              $f_{max} := f(M_s(p))$ ;
- 13              $p_{max} := p$
- 14          $Done := Done \cup c$ ;
- 15      $k := k + 1$ ;
- 16 **until**  $k == |\Phi|$  or timeout;
- 17 **return** bit-vector  $p_{max}$ ;

---

respectively denote its lower and upper safety thresholds. Let:

$$d_s = \begin{cases} \min(|v_s - L_s|, |v_s - H_s|) & L_s \leq v_s \leq H_s \\ 0 & \text{otherwise} \end{cases}$$

A suitable objective function that is maximised by values approaching either of the thresholds would then be:

$$f(v_s) = \frac{1}{d_s/(H_s - L_s)}$$

We calculate *feature importance* in one of two ways, depending on the model used. For a linear model, the absolute value of the model's weight for that feature is taken as its importance. For a GBDT model, since it is a boosting ensemble model with a bunch of decision trees, we average the feature importance scores of these trees to obtain the feature importance of the overall model.

For SWaT, we implemented attack discovery for multiple flow, pressure, and tank level sensors, and used instances of the objective function above for each of them. The feature vectors returned by Algorithm 3 are broken into their constituent packets, then spoofed in the network using Scapy. If an attack successfully drives a targeted sensor out of its normal operational range (e.g. over/underflow), we record this, adding the particular packet manipulation used to a test suite of attacks, and document it accordingly (see Section 4 for an experimental evaluation). Recall that in SWaT, the models for tank levels sensors do not predict future values directly, but rather the magnitude by which they will change by: as a consequence, Algorithm 3 is adapted for these sensors by observing the current reading at the beginning, then using it to calculate the input for the objective function.

## 4 EVALUATION

We evaluate the effectiveness of active fuzzing for attack discovery and detection using the SWaT testbed (Section 2).

### 4.1 Research Questions

Our evaluation design is centred around the following key research questions (RQs):

- RQ1 (Training Time):** How much time is required to learn a high-accuracy model?
- RQ2 (Attack Discovery):** Which model and active learning setup is most effective for attack discovery?
- RQ3 (Comparisons):** How does active fuzzing compare against other CPS fuzzing approaches?
- RQ4 (Attack Detection):** Can the learnt models be used for anomaly detection or early warnings?

RQ1 is motivated by our assumption that attackers do not have access to large offline datasets for training, and may need to evade anomaly detection systems. How long would an attacker need to spend observing live sensor readings (pre-training) and spoofing packets (active learning) before obtaining a high-accuracy model? RQ2 aims to explore the different combinations of our regression models with and without active learning, in order to establish which is most effective for discovering packet-level CPS attacks, and to quantify any added benefit of active learning in conquering the huge search space. RQ3 is intended to check our work against a baseline, i.e. its effectiveness in comparison to random search and another guided CPS network fuzzer. Finally, RQ4 aims to explore whether our learnt models can have a secondary application as part of an anomaly detection or early warning system for attacks.

### 4.2 Experiments and Discussion

We present the design of our experiments for each of the RQs in turn, as well as some tables of results and the conclusions we draw from them. The programs we developed to implement these experiments on the SWaT testbed are all available online [5].

**RQ1 (Training Time).** Our first RQ aims to assess the amount of time an attacker would require to learn a high-accuracy model from live packets. To answer this question, we design experiments for the two phases of learning in turn.

First, we investigate how long the attacker must spend *pre-training* on normal live sensor readings (i.e. without any manipulation). Recall that our goal in this phase is not to obtain a highly accurate model, but rather to find a *reasonable* enough model as a starting point for active learning. To do this, we compute the *r2 scores* of linear and GBDT regression models for individual sensors after training for different lengths of time. An *r2 score* is the percentage of variation explained by the model, and reflects how well correlated the predictions of a sensor and their actual future values are. Prior to training, we collect 230 minutes of packet and sensor data, splitting 180 minutes of it into a training set and the remaining 50 minutes into a test set. For each sensor, we train linear and GBDT models using the full 180 minutes (our upper limit of the experiment), and compute their *r2 scores* using the test data. We repeat this process for 10 minutes of data, then 20 minutes, ... up to 150 minutes at various intervals until it is clear that model is converging. We judge that a model has converged when the importance scores of its features (see Section 3.2) have stabilised up to a small tolerance (0.5% for flow/pressure sensors; 5% for level sensors) as the model is re-trained on new samples. All steps are

repeated ten times and medians are reported to reduce the effects of different starting states.

Second, given a model that has been pre-trained, we investigate how long the attacker must then spend *actively learning* before the model achieves a high accuracy. To do this, we pre-train linear and GBDT prediction models for each sensor for the minimum amount of time previously determined (in the first experiment). Then, for both variants of active learning (Section 3.2), we sample new sensor data from the system and retrain the models every 5 minutes. In this experiment, we record the amount of time it takes for a model to stabilise with a high r<sub>2</sub> score, i.e. above 0.9, using the same 50 minutes of test data to compute this. We repeat these steps ten times and compute the medians.

*Results.* Table 1 presents the results of our first experiment. The columns correspond to the amount of training time (10 minutes through to 180), whereas the rows correspond to regression models for individual SWaT sensors, including Flow Indicator Transmitters (e.g. FIT101), a Differential Pressure Indicator Transmitter (DPIT301), and Level Indicator Sensors (e.g. LIT101). For the LITs, our models predict their values 30 seconds into the future (as tank levels rise very slowly), whereas for all other sensors our models make predictions for 5 seconds ahead. The values reported in the table are r<sub>2</sub> scores: here, a score of 1 indicates that the model and test data are perfectly correlated, whereas a score of -1 would indicate that there is no correlation at all. When there is clear evidence of a model converging, we do not repeat the experiment for longer training periods (except 180 minutes, our upper limit).

All of our models eventually converge during pre-training, except the linear model for LIT401: the process involving this tank is too complicated to be represented as a linear model due to the multiple interactions and dependencies involving other stages of the testbed (the GBDT model does not suffer this problem). Note that while pre-training leads to relatively high r<sub>2</sub> scores for a number of the models (e.g. the simpler processes involving flow), this does not necessarily imply that the models will be effective for attack discovery (as we investigate in RQ2). For the goal of determining a minimum amount of pre-training time, we fix it at 40 minutes, as all models (except Linear-LIT401) exhibit some positive correlation by then ( $\geq 0.3$ ). Some scores are still low, but this will allow us to assess whether active learning is still effective when applied to cases that lack a good pre-trained model.

Table 2 presents the results of our second experiment. Here, the columns contain the sensors that each regression model is targeting, whereas the rows contain the type of model and active learning variant considered. The values reported are the number of minutes (accurate up to 5 minutes) of active learning that it takes before models achieve an r<sub>2</sub> score above 0.9. Note that with active learning, none of the linear models for tank level sensors were able to exceed our r<sub>2</sub> threshold (although they did converge for LIT101 and LIT301 with lower scores). All GBDT models were able to exceed the r<sub>2</sub> threshold with active learning, indicating that the additional expressiveness is important for some processes of SWaT—likely because the actual processes are non-linear. The amount of time required varied from 10 up to 45 minutes. Taking the pre-training time into consideration:

*Once pre-trained on 40 minutes of data observations, attackers can accurately predict SWaT's sensor readings after 10–45 minutes of active learning.*

This is a significantly reduced amount of time compared to SWaT's LSTM-based fuzzer [26], the model of which was trained for approximately two days on a rich dataset compiled from four days of constant operation.

**RQ2 (Attack Discovery).** Our second RQ aims to assess which combinations of models and active learning setups (including no active learning at all) are most effective for *finding attacks*, i.e. manipulations of packet payloads that would cause the true readings of a particular sensor to eventually cross one of its safety thresholds (e.g. risk of overflow, or risk of bursting pipe).

To do this, we experimentally calculate the *success rates* at finding attacks for all variants of models covering the flow, pressure, and tank level sensors. Furthermore, we do so while restricting the manipulation of the packets' payloads to different quantities of bit flips, from 1–5 and 10 such flips. For each model variant, we calculate the success rate by running our active fuzzer 1000 times with the given model, and recording as a percentage the number of times in which the resulting modified packet would cause the physical state to cross<sup>1</sup> a safety threshold. Note that it is important to flip existing payload bits, rather than craft packets directly, as the system's built-in validation procedures may reject them.

*Results.* Table 3 presents the results of our experiment for RQ2. Each sub-table reports on a restriction to a particular number of payload bit flips, ranging from 1–5 and then 10. The columns contain the sensors we are attempting to drive into unsafe states, whereas the rows contain the type of model and active learning variant considered (if any). The final row, Random (No Model), is discussed as part of RQ3. The values recorded are success rates (%), where 100% indicates that all 1000 model-guided bit flips would succeed, and where 0% indicates that none of them would do. In the active learning models, pre-training was conducted for 40 minutes (as determined in RQ1). We also include a model that was pre-trained *only* for 90 minutes—roughly the time to do *both* pre-training and active learning—to ensure a fair comparison.

We can draw a number of conclusions from these results. First, linear models are not expressive enough in general for driving the bit flipping of payloads: their success rates for the LITs, for example, is mostly 0%, and even at 10 bit flips numbers for most sensors remain very low. GBDT quite consistently outperforms the linear models, often approaching 100% success rates. Like the linear models, GBDT struggled to attack the LITs for small numbers of bit flips (likely because multiple commands are needed to affect these sensors), but can attack them all once the restriction is lifted to ten bit flips.

*The expressiveness of the underlying model is critically important: active learning alone is not enough to compensate for this.*

Another conclusion that can be drawn from the tables relates to the significantly higher success rates for variants using active learning, both for linear models and GBDT models. The combination of active learning with the expressiveness of GBDT, in particular, leads

<sup>1</sup>With the exception of the low threshold for flow sensors, which is 0.

**Table 1: r2 scores (*higher is better*) of linear and GBDT sensor prediction models after different amounts of pre-training**

Linear Models		10min	20min	30min	40min	50min	60min	70min	80min	100min	120min	150min	180min
Flow	FIT101	0.3399	0.5998	0.6698	0.7391	0.8214	0.8475	0.8896	0.8712	...	...	...	0.8840
	FIT201	-0.7112	-0.0775	0.7394	0.8381	0.8962	0.8931	...	...	...	...	...	0.7332
	FIT301	-0.481	0.5292	0.8227	0.8949	0.9121	0.9001	...	...	...	...	...	0.8772
	FIT401	-1.2513	-0.6149	0.1123	0.3142	0.6634	0.7235	0.6695	0.6143	...	...	...	0.6425
Pr.	DPIT301	-0.2511	0.6070	0.8648	0.9563	0.9651	0.9642	...	...	...	...	...	0.9569
T. Level	LIT101	0.0624	0.1516	0.6024	0.6582	0.6824	0.6168	0.7172	0.772	0.7965	0.8197	0.8133	0.8254
	LIT301	-0.0806	0.0937	0.4248	0.4949	0.5963	0.6260	0.6583	0.4807	0.6209	...	...	0.5426
	LIT401	0.1543	0.0612	0.2942	0.0273	-0.2208	0.1119	-0.4902	0.007	0.1259	0.2412	0.0135	-0.6597
GBDT Models		10min	20min	30min	40min	50min	60min	70min	80min	100min	120min	150min	180min
Flow	FIT101	-0.2229	-0.0245	0.4313	0.7742	0.9112	0.9413	0.9755	0.9741	...	...	...	0.9637
	FIT201	-0.7116	0.3545	0.9584	0.9633	0.9504	0.9642	...	...	...	...	...	0.9421
	FIT301	-0.9453	0.2496	0.8051	0.9734	0.9731	0.9751	...	...	...	...	...	0.9524
	FIT401	-0.2124	0.3120	0.7015	0.8125	0.8342	0.8861	0.8453	0.7921	...	...	...	0.8025
Pr.	DPIT301	-0.5508	0.8218	0.9387	0.9757	0.9818	0.9831	0.9912	0.9901	0.9875	0.9706	0.9496	0.9085
T. Level	LIT101	-0.042	-0.1352	-0.153	0.3858	0.6459	0.7881	0.8536	0.8680	0.8961	0.9221	0.8721	0.9019
	LIT301	-0.0419	-0.2151	-0.0185	0.3863	0.7059	0.8208	0.6486	0.7938	0.5498	0.7363	...	0.7291
	LIT401	-1.1415	0.1121	0.7123	0.8377	0.8503	0.8575	0.7731	0.7907	0.8764	0.8743	0.8741	0.8444

**Table 2: Median time (mins; *lower is better*) for active learning (AL) configurations to achieve an r2 score above 0.9**

AL Config.	Flow			Pressure		Tank Level		
	FIT101	FIT201	FIT301	FIT401	DPIT301	LIT101	LIT301	LIT401
Linear (EBCM)	25	15	30	15	30	—	—	—
Linear (EMCM)	20	20	45	15	40	—	—	—
GBDT (EBCM)	10	10	25	10	30	35	30	45
GBDT (EMCM)	10	10	25	10	20	40	40	45

to attacks being found in all cases for the 10 bit flip restriction. With active learning enabled, the difference is often significant (e.g. 0% vs. 100% for FIT401, 10 bit flips). The results suggest that active learning is key for finding the ‘critical bits’ in payloads, given its ability to sample and query new data. Models that have only been pre-trained just recognise trends observed in normal data, and do not necessarily know which bits involved in the patterns are the critical ones for enacting an attack.

Active learning is effective at identifying critical bits in payloads, and can lead to significantly higher success rates in attack discovery.

**RQ3 (Comparisons).** Our third RQ assesses how active fuzzing performs against two baselines. First, for every sensor (as targeted in RQ2), we randomly generate 1000  $k$ -bit payload manipulations (where  $k$  is 1–5 or 10) and assess for them the attack success rates (a percentage, as calculated in RQ2). Second, we qualitatively compare our attacks against the ones identified by the LSTM-based fuzzer for SWaT [26] as well as an established benchmark of SWaT network attacks [41] that was manually crafted by experts.

**Results.** The results of the random flipping baseline are given in the final rows of Table 3. Clearly, this is not an effective strategy for finding attacks based on packet manipulation, as no success rate exceeds 0.5%. This is unsurprising due to the huge search space involved. Note also that for the more challenging over/underflow attacks, random bit flipping is unable to find any examples at all.

Regarding the LSTM-based fuzzer for SWaT [26], a side-by-side comparison is difficult to make as it does not manipulate packets

but rather only issues high-level actuator commands (e.g. “OPEN MV101”). Our approach is able to find attacks spanning the same range of sensed properties, but does so by manipulating the bits of packets directly (closer to the likely behaviour of a real attacker) and without the same level of network control (other than true sensor readings, which both approaches require). In this sense our attacks are more elaborate than those of the LSTM fuzzer. Our approach is also substantially faster: active fuzzing can train effective models in 50–85 minutes, whereas the underlying model used in [26] required approximately two whole days.

Our coverage of the SWaT benchmark [41] is comparable to that of [26], since both approaches find attacks spanning the same sensed properties. However, all of the attacks in [41] and [26] are implemented at Level 1 of the network. Active fuzzing instead generates packet-manipulating attacks at Level 0, which has the advantage of avoiding interactions with the PLC code, possibly making manipulations harder to detect (e.g. bypassing command validation checks). In this sense, the attacks that active fuzzing finds complement and enrich the benchmark.

Active fuzzing finds attacks covering the same sensors as comparable work, but with significantly less training time, and by manipulating packets directly.

**RQ4 (Attack Detection).** Our final RQ considers whether our learnt models can be used not only for attack discovery but also attack *prevention*. In particular, we investigate their use in two defence mechanisms: an anomaly detector and an early warning system. We then assess how effective they are detecting attacks.

To perform anomaly detection, we continuously perform the following process: we read the current values of sensors, and then use our learnt models to predict their values 5 seconds into the future (30 seconds for tank levels). After 5 (or 30) seconds have passed, the *actual* values  $v_a$  are compared with those that were predicted  $v_p$ , and an anomaly is reported if  $|v_p - v_a|/v_m > 0.05$  (or  $|v_p - v_a| > 5$  for tanks), where  $v_m$  is the largest possible observable value for the sensor. To evaluate the effectiveness of this detection scheme, we implement an experiment on actual historian data extracted from SWaT [41]. For each sensor in turn, we randomly generate 1000 spoofed sensor values by randomly adding or subtracting values (in

**Table 3: Success rates (%) of different model configurations for finding packet manipulations (1-5 and 10 bit flips) that successfully drive SWaT's flow, pressure, and tank level readings to safety thresholds**

		Flow				Pr.	Level		
		FT101	FT201	FT301	FT401	DPT301	LIT101	LIT301	LIT401
<b>Models (1 Bit Flip)</b>									
Linear	Pre-Train Only (40min)	0.5	4.4	1.3	1.8	0.9	0	0	0
	Pre-Train Only (90min)	0.8	3.8	1.4	0.2	2.5	0	0	0
	Active Learning (EBCM)	27	22.3	7.5	43.9	14.4	0	0	0
	Active Learning (EMCM)	29.4	19.2	7.9	36.6	9.1	0	0	0
GBDT	Pre-Train Only (40min)	0	59.3	0	0	0	0	0	0
	Pre-Train Only (90min)	0	57.7	30.3	0	0	0	0	0
	Active Learning (EBCM)	97.7	99.2	97.9	75.4	96.1	0	0	0
	Active Learning (EMCM)	97.6	99.4	97.6	13.5	95.3	0	0	0
—	Random (No Model)	0	0	0	0.2	0	0	0	0
<b>Models (2 Bit Flips)</b>									
Linear	Pre-Train Only (40min)	0.7	8	1.9	3.1	1.7	0.2	0	0
	Pre-Train Only (90min)	2.7	9.4	4.1	0.6	6.6	0	0	0
	Act. Learning (EBCM)	46.1	39.1	15.1	77.6	30.1	2	0	0
	Act. Learning (EMCM)	47.4	31.8	16.5	72.4	17.9	0.6	0	0
GBDT	Pre-Train Only (40min)	0	98.7	0	0	0	0	0	0
	Pre-Train Only (90min)	0.3	96	59	0	0	0	0	0
	Act. Learning (EBCM)	100	99.9	100	100	99.9	76.4	0	0
	Act. Learning (EMCM)	100	100	99.7	100	99.9	87.1	0	0
—	Random (No Model)	0.3	0	0	0.2	0	0	0	0
<b>Models (3 Bit Flips)</b>									
Linear	Pre-Train Only (40min)	1.2	10.6	3.5	3.5	3	0	0	0
	Pre-Train Only (90min)	4.2	12.5	5.5	0.5	8.2	0	0	0
	Act. Learning (EBCM)	58.6	50.6	25.7	91.7	37.2	4.2	0.1	0.1
	Act. Learning (EMCM)	60.4	45.1	21.4	88.7	24.2	2.7	0.3	0
GBDT	Pre-Train Only (40min)	0	100	0.1	0	0	0	0	0
	Pre-Train Only (90min)	1.1	100	80.6	0	0	0	0	0
	Act. Learning (EBCM)	100	100	100	100	100	97	8.1	23.7
	Act. Learning (EMCM)	100	100	99.7	100	100	99.2	3.7	32.3
—	Random (No Model)	0.1	0.2	0.1	0.1	0.3	0	0	0
<b>Models (4 Bit Flips)</b>									
Linear	Pre-Train Only (40min)	0.9	13.8	5.4	4.9	4.3	0	0	0
	Pre-Train Only (90min)	6.4	15.1	7.5	0.9	12.1	0	0	0
	Act. Learning (EBCM)	71.6	65	27.9	97.7	49.1	11.4	0.1	0.2
	Act. Learning (EMCM)	75.1	26.8	31.1	95.7	29.2	5.4	0.3	0
GBDT	Pre-Train Only (40min)	0	100	0.7	0	0	0	0	0
	Pre-Train Only (90min)	1.2	100	96.1	0	0	0	0	0
	Act. Learning (EBCM)	100	100	100	100	100	100	20.4	55.7
	Act. Learning (EMCM)	100	100	99.7	100	100	100	17.7	67.3
—	Random (No Model)	0.1	0.1	0.1	0.2	0.2	0	0	0
<b>Models (5 Bit Flips)</b>									
Linear	Pre-Train Only (40min)	1.7	19.3	6.2	5.7	4.3	0.4	0	0
	Pre-Train Only (90min)	6.6	18.4	8.7	1.1	13.5	0.1	0	0
	Act. Learning (EBCM)	78.3	71.6	35.4	99.5	56.6	16.3	0.5	0
	Act. Learning (EMCM)	81.5	62.6	36	97.9	39.3	9.3	0.5	0
GBDT	Pre-Train Only (40min)	0	100	3.1	0	0	0	0	0
	Pre-Train Only (90min)	3.1	100	99.8	0	0.1	0	0	0
	Act. Learning (EBCM)	100	100	100	100	100	100	29.7	76
	Act. Learning (EMCM)	100	100	99.7	100	100	100	34.9	82.2
—	Random (No Model)	0.2	0.4	0.1	0.1	0.2	0	0	0
<b>Models (10 Bit Flips)</b>									
Linear	Pre-Train Only (40min)	4.7	34.9	9.6	12.5	10.7	0.8	0	0
	Pre-Train Only (90min)	13	38.3	19.2	2.3	29.1	1	0	0
	Act. Learning (EBCM)	96.4	91	61.8	100	81.5	35.7	6.1	2.1
	Act. Learning (EMCM)	96.7	89.1	59.5	100	63	29.1	5.7	0
GBDT	Pre-Train Only (40min)	0	100	31.5	0	0	0	0	0
	Pre-Train Only (90min)	12.1	100	99.8	0	2.3	0	0	0
	Act. Learning (EBCM)	100	100	100	100	100	100	72.2	99.3
	Act. Learning (EMCM)	100	100	99.7	100	100	100	77	99.7
—	Random (No Model)	0.2	0.4	0.4	0.1	0.5	0	0	0

**Table 4: Success rates (%) of different anomaly detector models at detecting injected sensor values**

		Flow				Pr.	Tank Level		
		FT101	FT201	FT301	FT401	DPT301	LIT101	LIT301	LIT401
<b>Anomaly Detector</b>									
Linear	Pre-Train Only (40min)	82.3	100	100	41.3*	100	8.2*	38.5*	40.1*
	Pre-Train Only (90min)	100	100	100	62.9*	100	67.8*	39.3*	13.2*
	Act. Learning (EBCM)	100	100	100	71.1*	100	59.8*	51.2*	64.1*
	Act. Learning (EMCM)	100	100	100	71.4*	100	57.5*	44.7*	50.7*
GBDT	Pre-Train Only (40min)	100	100	100	100	100	71.8*	74.6*	76*
	Pre-Train Only (90min)	100	100	100	100	100	95.3	92.3	74.1
	Act. Learning (EBCM)	100	100	100	100	100	91.8	95.5	84.1
	Act. Learning (EMCM)	100	100	100	100	100	92.5	95.8	83.4

the range 5-10 for LITs, or  $0.05v_m$  through to  $0.1v_m$  for the others) to sensor readings at different points of the data. We then use our learnt models to determine what would have been predicted from the data 5 or 30 seconds earlier, comparing the actual and predicted values as described. We record the success rates of our anomaly detectors at detecting these spoofed sensor readings.

Our early warning system is set up in a similar way, continuously predicting the future readings of sensors based on the current network traffic. The key difference is that rather than comparing actual values with previously predicted values, we instead issue warnings at the time of prediction if the future value of a sensor is outside of its well-defined normal operational range. To experimentally assess this, we manually subject the SWaT testbed to the Level 1 attacks identified in [26] (which itself covers more unsafe states than the SWaT benchmark [41]), targeting each sensor in turn. When each attack is underway, we use our learnt models to predict the future sensor readings. If a warning is issued at some point *before* a sensor is driven outside of its normal range, we record this as a success. We repeat this ten times for each sensor.

*Results.* Table 4 contains the results of our anomaly detection experiment. The columns indicate the sensors for which values in the data were manipulated, whereas the rows indicate the model and active learning variant used. The values are the success rates, i.e. the percentage of spoofed sensor values that were detected as anomalous. Asterisks (\*) indicate where false positive rates were above 5%, meaning the anomaly detectors were not practically useful. For the flow and pressure sensors, most variants of model

**Table 5: Success rates (%)<sup>3</sup>; higher is better) of different models at warning before sensors exit their safe ranges**

Early Warning System		Tank Level		
		LIT101	LIT301	LIT401
GBDT	Linear (all variants)	—	—	—
	Pre-Train Only (40min)	—	—	—
	Pre-Train Only (90min)	100	100	100
	Active Learning (EBCM)	100	100	100
EMCM	Active Learning (EMCM)	100	100	100

and active learning were able to successfully detect anomalies, the main exception being FIT401 for which the linear model performed poorly. The tank level sensors were more challenging to perform anomaly detection for, but the GBDT models have a clear edge over the linear ones. Active learning made little difference across the experiments, except to improve the accuracy of the original 40 minute pre-trained models.

Table 5 contains the results of our early warning detection experiment. The columns indicate the sensed properties that were targeted by attacks (e.g. drive LIT101 outside of its safe range), whereas the rows indicate the model and active learning variant used. The values are the success rates, i.e. the percentages of attacks that were warned about before succeeding. Cells containing dashes (—) indicate that more than 5% of the warnings were false positive, and thus too unreliable. The first thing to note is that the experiment only considered the tank level sensors: this is because the flow and pressure sensors can be forced into unsafe states very quickly, requiring more immediate measures than an early warning system. The tanks however take time to fill up or empty, and thus are a more meaningful target for this solution. Second, the model has a clear impact: GBDT models with either active learning or at least 90 minutes of pre-training are accurate enough to warn about 100% of the attacks, whereas the linear models are not expressive enough and suffer from false positives. Again, active learning improves the accuracy of 40 minute pre-trained models sufficiently, but otherwise is not critical: its key role is not in prevention but in *discovering* attacks, through its ability to identify the critical bits to manipulate.

*Our models can be repurposed as anomaly detectors or early warning systems, but active learning is not as critical here as in attack discovery.*

### 4.3 Threats to Validity

While our work has been extensively evaluated on a real critical infrastructure testbed, threats to the validity of our conclusions of course remain. First, while SWaT is a fully operational water treatment testbed, it is not as large as the plants it is based on, meaning our results may not scale-up (this is difficult to assess, as access to such plants is subject to strict confidentiality). Second, it may not generalise to CPSs in domains that have different operational characteristics, motivating future work to properly assess this. Finally, while our anomaly detector performed well, our sensor spoofing attacks were generated randomly, and may not be representative of a real attacker's behaviour (note however that the early warning system was assessed using previously documented attacks). Similarly, our early warning detection systems performed well at detecting

known over/underflow attacks, but these attacks are of the kind that active fuzzing itself can generate: how the models perform against different kinds of attacks requires further investigation.

## 5 RELATED WORK

In this section, we highlight a selection of the literature that is related to the main themes of this paper: *learning from traffic* (including active learning), *defending CPSs*, and *testing/verifying CPSs*.

**Learning from Network Traffic.** The application of machine learning to network traffic is a vibrant area of research [68], but models are typically constructed to perform *classification* tasks. To highlight a few examples: Zhang et al. [84] combine supervised and unsupervised ML to learn models that can classify zero-day traffic; Nguyen and Armitage [67] learn from statistical features of sub-flows to classify between regular consumer traffic and traffic originating from online games; and Atkinson et al. [16] use a classifier to infer personal information by analysing encrypted traffic patterns caused by mobile app usage. All these examples are in contrast to active fuzzing, where *regression* models are learnt for predicting how a set of network packets will cause a (true) sensor reading of a CPS to change. We are not aware of other work building regression models in a similar context.

Similar to active fuzzing, there are some works that apply active learning, but again for the purpose of classification, rather than regression. Morgan [65], for example, uses active learning to reduce training time for streaming data classifiers, as do Zhao and Hoi [86] but for malicious URL classifiers.

**Defending CPSs.** Several different research directions on detecting and preventing CPS attacks have emerged in the last few years. Popular approaches include anomaly detection, where data logs (e.g. from historians) are analysed for suspicious events or patterns [11, 15, 27, 45, 48, 52, 58, 61, 66, 69, 71]; digital fingerprinting, where sensors are checked for spoofing by monitoring time and frequency domain features from sensor and process noise [12, 13, 44, 56]; and invariant-based checks, where conditions over processes and components are constantly monitored [7, 8, 10, 18, 24, 25, 28, 39]. These techniques are meant to complement and go beyond the built-in validation procedures installed in CPSs, which typically focus on simpler and more localised properties of the system.

The strengths and weaknesses of different countermeasures has been the focus of various studies. Urbina et al. [77] evaluated several attack detection mechanisms in a comprehensive review, concluding that many of them are not limiting the impact of stealthy attacks (i.e. from attackers who have knowledge about the system's defences), and suggest ways of mitigating this. Cárdenas et al. [19] propose a general framework for assessing attack detection mechanisms, but in contrast to the previous works, focus on the business cases between different solutions. For example, they consider the cost-benefit trade-offs and attack threats associated with different methods, e.g. centralised vs. distributed.

As a testbed dedicated for cyber-security research, many different countermeasures have been developed for SWaT itself. These include anomaly detectors, typically trained on the publicly released dataset [2, 41] using unsupervised learning techniques, e.g. [42, 52,

58]. A supervised learning approach is pursued by [24, 25], who inject faults into the PLC code of (a high-fidelity simulator) in order to obtain abnormal data for training. Ahmed et al. [12, 13] implemented fingerprinting systems based on sensor and process noise for detecting spoofing. Adepu and Mathur [7, 8, 10] systematically and manually derived physics-based invariants and other conditions to be monitored during the operation of SWaT. Feng et al. [32] also generate invariants, but use an approach based on learning and data mining that can capture noise in sensor measurements more easily than manual approaches.

**Testing and Verifying CPSs.** Several authors have sought to improve the defences of CPSs by constructing or synthesising attacks that demonstrate flaws to be fixed. Liu et al. [62] and Huang et al. [50], for example, synthesise attacks for power grids that can bypass bad measurement detection systems and other conventional monitors. Dash et al. [30] target robotic vehicles, which are typically protected using control-based monitors, and demonstrate three types of stealthy attacks that evade detection. Uluagac et al. [76] presented attacks on sensory channels (e.g. light, infrared, acoustic, and seismic), and used them to inform the design of an intrusion detection system for sensory channel threats. Active fuzzing shares this goal of identifying attacks in order to improve CPS defences.

Fuzzing is a popular technique for automatically testing the defences of systems, by providing them with invalid, unexpected, or random input and monitoring how they respond. Our active fuzzing approach does exactly this, guiding the construction of input (network packets) using prediction models, and then observing sensor readings to understand how the system responds. The closest fuzzing work to ours is [26], which uses an LSTM-based model to generate actuator configurations, but requires vast amounts of data and system access to function effectively. Fuzzing has also been applied for testing CPS models, e.g. CyFuzz [29] and DeepFuzzSL [72], which target models developed in Simulink. Outside of the CPS domain, several fuzzing tools are available for software: American fuzzy lop [83], for example, uses genetic algorithms to increase the code coverage of tests; Cha et al. [22] use white-box symbolic analysis on execution traces to maximise the number of bugs they find; and grammar-based fuzzers (e.g. [40, 49]) use formal grammars to generate complex structured input, such as HTML-/JavaScript for testing web browsers. Fuzzing can also be applied to network protocols in order to test their intrusion detection systems (e.g. [79]). Our work, in contrast, assumes that an attacker has *already compromised* the network (as per Section 2).

There are techniques beyond fuzzing available for analysing CPS models in Simulink. A number of authors have proposed automated approaches for falsifying such models, i.e. for finding counterexamples of formal properties. To achieve this, Yamagata et al. [14, 81] use deep reinforcement learning, and Silvetti et al. [73] use active learning. Chen et al. [23] also use active learning, but for mining formal requirements from CPS models. Note that unlike these approaches, active fuzzing is applied directly at the network packet level of a real and complex CPS, and therefore does not make any of the abstractions that modelling languages necessitate.

A number of approaches exist that allow for CPSs to be *formally* verified or analysed. These typically require a formal specification or model, which, if available in the first place, may abstract

away important complexities of full-fledged CPS processes. Kang et al. [55], for example, construct a discretised first-order model of SWaT's first three stages in Alloy, and analyse it with respect to some safety properties. This work, however, uses high-level abstractions of the physical process, only partially models the system, and would not generalise to the packet-level analyses that active fuzzing performs. Sugumar and Mathur [75] analyse CPSs using timed automata models, simulating their behaviour under single-stage single-point attacks. Castellanos et al. [20], McLaughlin et al. [64], and Zhang et al. [85] perform formal analyses based on models extracted from the PLC programs, whereas Etigowni et al. [31] analyse information flow using symbolic execution. If a CPS can be modelled as a hybrid system, then a number of formal techniques may be applied, including model checking [34, 80], SMT solving [36], reachability analysis [54], non-standard analysis [47], process calculi [59], concolic testing [57], and theorem proving [70]. Defining a formal model that accurately characterises enough of the CPS, however, is the *hardest* part, especially for techniques such as active fuzzing that operate directly at the level of packet payloads.

## 6 CONCLUSION

We proposed *active fuzzing*, a black-box approach for automatically building test suites of packet-level CPS network attacks, overcoming the enormous search spaces and resource costs of such systems. Our approach learnt regression models for predicting future sensor values from the binary string payloads of network packets, and used these models to identify payload manipulations that would achieve specific attack goals (i.e. pushing true sensor values outside of their safe operational ranges). Key to achieving this was our use of online active learning, which reduced the amount of training data needed by sampling examples that were estimated to maximally improve the model. We adapted the EMCM [17] active learning framework to CPSs, and proposed a new version of it that guided the process by maximising behaviour change.

We presented algorithms for implementing active fuzzing, but also demonstrated its efficacy by implementing it for the SWaT testbed, a multi-stage water purification plant involving complex physical and chemical processes. Our approach was able to achieve comparable coverage to an established benchmark and LSTM-based fuzzer, but with significantly less data, training time, and resource usage. Furthermore, this coverage was achieved by more sophisticated attacks than those of the LSTM-based fuzzer, which can only generate high-level actuator commands and is unable to manipulate packets directly. Finally, we showed that the models constructed in active learning were not only useful for attack *discovery*, but also for attack *detection*, by implementing them as anomaly detectors and early warning systems for SWaT. We subjected the plant to a series of random sensor-modification attacks as well as existing actuator-manipulation attacks, finding that our most expressive learnt models were effective at detecting them.

## ACKNOWLEDGMENTS

We are grateful to the three anonymous ISSTA referees for their very constructive feedback. This research / project is supported by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme “Design Science and Technology

for Secure Critical Infrastructure" (Award Number: NSoE\_DeST-SCI2019-0008). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. It is also supported in part by a Major Scientific Research Project of Zhejiang Lab (2018FD0ZX01), Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang Key R&D Plan (2019C03133), the Fundamental Research Funds for the Central Universities (2020QNA5021), and by an SUTD-ZJU IDEA Grant for Visiting Professor (SUTD-ZJU VP 201901).

## REFERENCES

- [1] 2016. CVE-2017-0144. Available from MITRE, CVE-ID CVE-2017-0144.. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144>
- [2] 2020. iTrust Labs: Datasets. [https://itrust.sutd.edu.sg/itrust-labs\\_datasets/](https://itrust.sutd.edu.sg/itrust-labs_datasets/). Accessed: May 2020.
- [3] 2020. Scapy. <https://scapy.net/>. Accessed: May 2020.
- [4] 2020. Secure Water Treatment (SWaT). [https://itrust.sutd.edu.sg/itrust-labs-home/itrust-labs\\_swat/](https://itrust.sutd.edu.sg/itrust-labs-home/itrust-labs_swat/). Accessed: May 2020.
- [5] 2020. Supplementary material. [https://github.com/yuqiChen94/Active\\_fuzzer](https://github.com/yuqiChen94/Active_fuzzer).
- [6] 2020. Wireshark. <https://www.wireshark.org/>. Accessed: May 2020.
- [7] Sridhar Adepu and Aditya Mathur. 2016. Distributed Detection of Single-Stage Multipoint Cyber Attacks in a Water Treatment Plant. In *Proc. ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016)*. ACM, 449–460.
- [8] Sridhar Adepu and Aditya Mathur. 2016. Using Process Invariants to Detect Cyber Attacks on a Water Treatment System. In *Proc. International Conference on ICT Systems Security and Privacy Protection (SEC 2016) (IFIP AICT)*, Vol. 471. Springer, 91–104.
- [9] Sridhar Adepu and Aditya Mathur. 2018. Assessing the Effectiveness of Attack Detection at a Hackfest on Industrial Control Systems. *IEEE Transactions on Sustainable Computing* (2018).
- [10] Sridhar Adepu and Aditya Mathur. 2018. Distributed Attack Detection in a Water Treatment Plant: Method and Case Study. *IEEE Transactions on Dependable and Secure Computing* (2018).
- [11] Ekta Aggarwal, Mehdi Karimibiki, Karthik Patabiraman, and André Ivanov. 2018. CORGIDS: A Correlation-based Generic Intrusion Detection System. In *Proc. Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC 2018)*. ACM, 24–35.
- [12] Chuadhy Mujeeb Ahmed, Martín Ochoa, Jianying Zhou, Aditya P. Mathur, Rizwan Qadeer, Carlos Murguia, and Justin Ruths. 2018. *NoisePrint*: Attack Detection Using Sensor and Process Noise Fingerprint in Cyber Physical Systems. In *Proc. Asia Conference on Computer and Communications Security (AsiaCCS 2018)*. ACM, 483–497.
- [13] Chuadhy Mujeeb Ahmed, Jianying Zhou, and Aditya P. Mathur. 2018. Noise Matters: Using Sensor and Process Noise Fingerprint to Detect Stealthy Cyber Attacks and Authenticate sensors in CPS. In *Proc. Annual Computer Security Applications Conference (ACSAC 2018)*. ACM, 566–581.
- [14] Takumi Akazaki, Shuang Liu, Yoriyuki Yamagata, Yihai Duan, and Jianye Hao. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *Proc. International Symposium on Formal Methods (FM 2018) (LNCS)*, Vol. 10951. Springer, 456–465.
- [15] Wissam Aoudi, Mikel Iturbe, and Magnus Almgren. 2018. Truth Will Out: Departure-Based Process-Level Detection of Stealthy Attacks on Control Systems. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 817–831.
- [16] John S. Atkinson, John E. Mitchell, Miguel Rio, and George Maticch. 2018. Your WiFi is leaking: What do your mobile apps gossip about you? *Future Generation Comp. Syst.* 80 (2018), 546–557.
- [17] Wenbin Cai, Ya Zhang, and Jun Zhou. 2013. Maximizing Expected Model Change for Active Learning in Regression. In *Proc. IEEE 13th International Conference on Data Mining (ICDM 2013)*. IEEE Computer Society, 51–60.
- [18] Alvaro A. Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. 2011. Attacks against process control systems: risk assessment, detection, and response. In *Proc. ACM Symposium on Information, Computer and Communications Security (AsiaCCS 2011)*. ACM, 355–366.
- [19] Alvaro A. Cárdenas, Robin Berthier, Rakesh B. Bobba, Jun Ho Huh, Jorjeta G. Jetcheva, David Grochocki, and William H. Sanders. 2014. A Framework for Evaluating Intrusion Detection Architectures in Advanced Metering Infrastructures. *IEEE Transactions on Smart Grid* 5, 2 (2014), 906–915.
- [20] John H. Castellanos, Martín Ochoa, and Jianying Zhou. 2018. Finding Dependencies between Cyber-Physical Domains for Security Testing of Industrial Control Systems. In *Proc. Annual Computer Security Applications Conference (ACSAC 2018)*. ACM, 582–594.
- [21] Rui M. Castro, Rebecca Willett, and Robert D. Nowak. 2005. Faster Rates in Regression via Active Learning. In *Proc. Annual Conference on Neural Information Processing Systems (NIPS 2005)*. 179–186.
- [22] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proc. IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE Computer Society, 725–741.
- [23] Gang Chen, Zachary Sabato, and Zhaodan Kong. 2016. Active learning based requirement mining for cyber-physical systems. In *Proc. IEEE Conference on Decision and Control (CDC 2016)*. IEEE, 4586–4593.
- [24] Yuqi Chen, Christopher M. Poskitt, and Jun Sun. 2016. Towards Learning and Verifying Invariants of Cyber-Physical Systems by Code Mutation. In *Proc. International Symposium on Formal Methods (FM 2016) (LNCS)*, Vol. 9995. Springer, 155–163.
- [25] Yuqi Chen, Christopher M. Poskitt, and Jun Sun. 2018. Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System. In *Proc. IEEE Symposium on Security and Privacy (S&P 2018)*. IEEE Computer Society, 648–660.
- [26] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE Computer Society, 962–973.
- [27] Long Cheng, Ke Tian, and Danfeng (Daphne) Yao. 2017. Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. In *Proc. Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, 315–326.
- [28] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Xinyan. 2018. Detecting Attacks Against Robotic Vehicles: A Control Invariant Approach. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 801–816.
- [29] Shaiful Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. 2017. CyFuzz: A Differential Testing Framework for Cyber-Physical Systems Development Environments. In *Proc. Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy 2016) (LNCS)*, Vol. 10107. Springer, 46–60.
- [30] Pritam Dash, Mehdi Karimibiki, and Karthik Patabiraman. 2019. Out of control: stealthy attacks against robotic vehicles protected by control-based techniques. In *Proc. Annual Computer Security Applications Conference (ACSAC 2019)*. ACM, 660–672.
- [31] Sriharsha Etigowni, Dave (Jing) Tian, Grant Hernandez, Saman A. Zonouz, and Kevin R. B. Butler. 2016. CPAC: securing critical infrastructure with cyber-physical access control. In *Proc. Annual Conference on Computer Security Applications (ACSAC 2016)*. ACM, 139–152.
- [32] Cheng Feng, Venkata Reddy Palleli, Aditya Mathur, and Deepthi Chana. 2019. A Systematic Framework to Generate Invariants for Anomaly Detection in Industrial Control Systems. In *Proc. Annual Network and Distributed System Security Symposium (NDSS 2019)*. The Internet Society.
- [33] Shai Fine, Ran Gilad-Bachrach, and Eli Shamir. 2002. Query by committee, linear separation and random walks. *Theoretical Computer Science* 284, 1 (2002), 25–51.
- [34] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. International Conference on Computer Aided Verification (CAV 2011) (LNCS)*, Vol. 6806. Springer, 379–395.
- [35] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics* 29 (2001), 1189–1232.
- [36] Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Proc. International Conference on Automated Deduction (CADE 2013) (LNCS)*, Vol. 7898. Springer, 208–214.
- [37] Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. 2003. *Kernel Query By Committee (KQBC)*. Technical Report. Leibniz Center, The Hebrew University.
- [38] Ran Gilad-Bachrach, Amir Navot, and Naftali Tishby. 2005. Query by Committee Made Real. In *Proc. Annual Conference on Neural Information Processing Systems (NIPS 2005)*. 443–450.
- [39] Jairo Giraldo, David I. Urbina, Alvaro Cardenas, Junia Valente, Mustafa Amir Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candel. 2018. A Survey of Physics-Based Attack Detection in Cyber-Physical Systems. *Comput. Surveys* 51, 4 (2018), 76:1–76:36.
- [40] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: machine learning for input fuzzing. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Computer Society, 50–59.
- [41] Jonathan Goh, Sridhar Adepu, Khurum Nazir Junejo, and Aditya Mathur. 2016. A Dataset to Support Research in the Design of Secure Water Treatment Systems. In *Proc. International Conference on Critical Information Infrastructures Security (CRITIS 2016)*.
- [42] Jonathan Goh, Sridhar Adepu, Marcus Tan, and Zi Shan Lee. 2017. Anomaly detection in cyber physical systems using recurrent neural networks. In *Proc. International Symposium on High Assurance Systems Engineering (HASE 2017)*. IEEE, 140–145.
- [43] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

- [44] Qinchen Gu, David Formby, Shouling Ji, Hasan Cam, and Raheem A. Beyah. 2018. Fingerprinting for Cyber-Physical System Security: Device Physics Matters Too. *IEEE Security & Privacy* 16, 5 (2018), 49–59.
- [45] Yoshiyuki Harada, Yoriyuki Yamagata, Osamu Mizuno, and Eun-Hye Choi. 2017. Log-Based Anomaly Detection of CPS Using a Statistical Method. In *Proc. International Workshop on Empirical Software Engineering in Practice (IWESEP 2017)*. IEEE, 1–6.
- [46] Amin Hassanzadeh, Amin Rasekh, Stefano Galelli, Mohsen Aghashahi, Riccardo Taormina, Avi Ostfeld, and M. Katherine Banks. 2019. A Review of Cybersecurity Incidents in the Water Sector. *Journal of Environmental Engineering* (09 2019).
- [47] Ichiro Hasuo and Kohei Suenaga. 2012. Exercises in Nonstandard Static Analysis of Hybrid Systems. In *Proc. International Conference on Computer Aided Verification (CAV 2012) (LNCS)*, Vol. 7358. Springer, 462–478.
- [48] Zecheng He, Aswin Raghavan, Guangyuan Hu, Sek M. Chai, and Ruby B. Lee. 2019. Power-Grid Controller Anomaly Detection with Enhanced Temporal Deep Learning. In *Proc. IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2019)*. IEEE, 160–167.
- [49] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. USENIX Security Symposium (USENIX 2012)*. USENIX Association, 445–458.
- [50] Zhenqi Huang, Sriharsha Etigowni, Luis Garcia, Sayan Mitra, and Saman A. Zonouz. 2018. Algorithmic Attack Synthesis Using Hybrid Dynamics of Power Grid Critical Infrastructures. In *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2018)*. IEEE Computer Society, 151–162.
- [51] ICS-CERT Alert. 2016. Cyber-Attack Against Ukrainian Critical Infrastructure. <https://ics-cert.us-cert.gov/alerts/IR ALERT-H-16-056-01>. document number: IR-ALERT-H-16-056-01.
- [52] Jun Inoue, Yoriyuki Yamagata, Yuqi Chen, Christopher M. Poskitt, and Jun Sun. 2017. Anomaly Detection for a Water Treatment System Using Unsupervised Machine Learning. In *Proc. IEEE International Conference on Data Mining Workshops (ICDMW 2017): Data Mining for Cyberphysical and Industrial Systems (DMCIS 2017)*. IEEE, 1058–1065.
- [53] ISA. 2020. ISA99, Industrial Automation and Control Systems Security. <https://www.isa.org/isa99/>. Accessed: May 2020.
- [54] Taylor T. Johnson, Stanley Bak, Marco Caccamo, and Lui Sha. 2016. Real-Time Reachability for Verified Simplex Design. *ACM Transactions on Embedded Computing Systems* 15, 2 (2016), 26:1–26:27.
- [55] Eunsuk Kang, Sridhar Adepu, Daniel Jackson, and Aditya P. Mathur. 2016. Model-based security analysis of a water treatment system. In *Proc. International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS 2016)*. ACM, 22–28.
- [56] Marcel Kneib and Christopher Huth. 2018. Scission: Signal Characteristic-Based Sender Identification and Intrusion Detection in Automotive Networks. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. ACM, 787–800.
- [57] Pingfan Kong, Yi Li, Xiaohong Chen, Jun Sun, Meng Sun, and Jingyi Wang. 2016. Towards Concolic Testing for Hybrid Systems. In *Proc. International Symposium on Formal Methods (FM 2016) (LNCS)*, Vol. 9995. Springer, 460–478.
- [58] Moshe Kravchik and Asaf Shabtai. 2018. Detecting Cyber Attacks in Industrial Control Systems Using Convolutional Neural Networks. In *Proc. Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC 2018)*. ACM, 72–83.
- [59] Ruggero Lanotte, Massimo Merro, Riccardo Muradore, and Luca Viganò. 2017. A Formal Approach to Cyber-Physical Attacks. In *Proc. IEEE Computer Security Foundations Symposium (CSF 2017)*. IEEE Computer Society, 436–450.
- [60] John Leyden. 2016. Water treatment plant hacked, chemical mix changed for tap supplies. *The Register* (2016). [https://www.theregister.co.uk/2016/03/24/water\\_utility\\_hacked/](https://www.theregister.co.uk/2016/03/24/water_utility_hacked/) Accessed: May 2020.
- [61] Qin Lin, Sridhar Adepu, Sicco Verwer, and Aditya Mathur. 2018. TABOR: A Graphical Model-based Approach for Anomaly Detection in Industrial Control Systems. In *Proc. Asia Conference on Computer and Communications Security (AsiaCCS 2018)*. ACM, 525–536.
- [62] Yao Liu, Peng Ning, and Michael K. Reiter. 2011. False data injection attacks against state estimation in electric power grids. *ACM Transactions on Information and System Security* 14, 1 (2011), 13:1–13:33.
- [63] Edwin Lughofer. 2017. On-line active learning: A new paradigm to improve practical usability of data stream modeling methods. *Information Sciences* 415 (2017), 356–376.
- [64] Stephen E. McLaughlin, Saman A. Zonouz, Devin J. Pohly, and Patrick D. McDaniel. 2014. A Trusted Safety Verifier for Process Controller Code. In *Proc. Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society.
- [65] Jillian Morgan. 2015. *Streaming Network Traffic Analysis Using Active Learning*. Ph.D. Dissertation. Dalhousie University.
- [66] Vedanth Narayanan and Rakesh B. Bobba. 2018. Learning Based Anomaly Detection for Industrial Arm Applications. In *Proc. Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC 2018)*. ACM, 13–23.
- [67] Thuy T. Nguyen and Grenville J. Armitage. 2006. Training on multiple sub-flows to optimise the use of Machine Learning classifiers in real-world IP networks. In *Proc. Annual IEEE Conference on Local Computer Networks (LCN 2006)*. IEEE Computer Society, 369–376.
- [68] Thuy T. T. Nguyen and Grenville J. Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials* 10, 1–4 (2008), 56–76.
- [69] Fabio Pasqualetti, Florian Dorfler, and Francesco Bullo. 2011. Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design. In *Proc. IEEE Conference on Decision and Control and European Control Conference (CDC-ECC 2011)*. IEEE, 2195–2201.
- [70] Jan-David Quesel, Stefan Mitsch, Sarah M. Loos, Nikos Arechiga, and André Platzer. 2016. How to model and prove hybrid systems with KeYmaera: a tutorial on safety. *International Journal on Software Tools for Technology Transfer* 18, 1 (2016), 67–91.
- [71] Peter Schneider and Konstantin Böttinger. 2018. High-Performance Unsupervised Anomaly Detection for Cyber-Physical System Networks. In *Proc. Workshop on Cyber-Physical Systems Security and PrivaCy (CPS-SPC 2018)*. ACM, 1–12.
- [72] Sohil Lal Shrestha, Shafiu Azam Chowdhury, and Christoph Csallner. 2020. DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain. In *Proc. Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest 2020)*. ACM. To appear.
- [73] Simone Silvetti, Alberto Policriti, and Luca Bortolussi. 2017. An Active Learning Approach to the Falsification of Black Box Cyber-Physical Systems. In *Proc. International Conference on integrated Formal Methods (iFM 2017) (LNCS)*, Vol. 10510. Springer, 3–17.
- [74] Chad Spensky, Aravind Machiry, Marcel Busch, Kevin Leach, Rick Housley, Christopher Kruegel, and Giovanni Vigna. 2020. TRUST.IO: Protecting Physical Interfaces on Cyber-physical Systems. In *Proc. IEEE Conference on Communications and Network Security (CNS 2020)*. IEEE. To appear.
- [75] Gayathri Sugumar and Aditya Mathur. 2017. Testing the Effectiveness of Attack Detection Mechanisms in Industrial Control Systems. In *Proc. IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C 2017)*. IEEE, 138–145.
- [76] A. Selcuk Uluagac, Venkatachalam Subramanian, and Raheem A. Beyah. 2014. Sensory channel threats to Cyber Physical Systems: A wake-up call. In *Proc. IEEE Conference on Communications and Network Security (CNS 2014)*. IEEE, 301–309.
- [77] David I. Urbina, Jairo Alonso Giraldo, Alvaro A. Cárdenas, Nilo Ole Tippenhauer, Junia Valente, Mustafa Amir Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. 2016. Limiting the Impact of Stealthy Attacks on Industrial Control Systems. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*. ACM, 1092–1105.
- [78] US National Science Foundation. 2018. Cyber-Physical Systems (CPS). [https://www.nsf.gov/publications/pub\\_summ.jsp?ods\\_key=nsf18538&org=NSF](https://www.nsf.gov/publications/pub_summ.jsp?ods_key=nsf18538&org=NSF). document number: nsf18538.
- [79] Giovanni Vigna, William K. Robertson, and Davide Balzarotti. 2004. Testing network-based intrusion detection signatures using mutant exploits. In *Proc. ACM Conference on Computer and Communications Security (CCS 2004)*. ACM, 21–30.
- [80] Jingyi Wang, Jun Sun, Yifan Jia, Shengchao Qin, and Zhiwu Xu. 2018. Towards ‘Verifying’ a Water Treatment System. In *Proc. International Symposium on Formal Methods (FM 2018) (LNCS)*, Vol. 10951. Springer, 73–92.
- [81] Yoriyuki Yamagata, Shuang Liu, Takumi Akazaki, Yihai Duan, and Jianye Hao. 2020. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. *IEEE Transactions on Software Engineering* (2020). Early access.
- [82] Cheah Huei Yoong, Venkata Reddy Palletti, Arlindo Silva, and Christopher M. Poskitt. 2020. Towards Systematically Deriving Defence Mechanisms from Functional Requirements of Cyber-Physical Systems. In *Proc. ACM Cyber-Physical System Security Workshop (CPSS 2020)*. ACM. To appear.
- [83] Michał Zalewski. 2017. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>. Accessed: May 2020.
- [84] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. 2015. Robust Network Traffic Classification. *IEEE/ACM Transactions on Networking* 23, 4 (2015), 1257–1270.
- [85] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James R. Moyne, and Z. Morley Mao. 2019. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *Proc. IEEE Symposium on Security and Privacy (S&P 2019)*. IEEE, 522–538.
- [86] Peilin Zhao and Steven C. H. Hoi. 2013. Cost-sensitive online active learning with application to malicious URL detection. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2013)*. ACM, 919–927.



# Learning Input Tokens for Effective Fuzzing

Björn Mathis

CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany  
bjoern.mathis@cispa.saarland

Rahul Gopinath

CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany  
rahul.gopinath@cispa.saarland

Andreas Zeller

CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany  
zeller@cispa.saarland

## ABSTRACT

Modern fuzzing tools like AFL operate at a *lexical* level: They explore the input space of tested programs one byte after another. For inputs with complex *syntactical* properties, this is very inefficient, as keywords and other tokens have to be composed one character at a time. Fuzzers thus allow to specify *dictionaries* listing possible tokens the input can be composed from; such dictionaries speed up fuzzers dramatically. Also, fuzzers make use of dynamic tainting to track input tokens and infer values that are expected in the input validation phase. Unfortunately, such tokens are usually implicitly converted to program specific values which causes a loss of the taints attached to the input data in the lexical phase.

In this paper, we present a technique to extend dynamic tainting to not only track explicit data flows but also taint implicitly converted data without suffering from taint explosion. This extension makes it possible to augment existing techniques and automatically infer a set of tokens and seed inputs for the input language of a program given nothing but the source code. Specifically targeting the lexical analysis of an input processor, our lFUZZER test generator systematically explores branches of the lexical analysis, producing a set of tokens that fully cover all decisions seen. The resulting set of tokens can be directly used as a dictionary for fuzzing. Along with the token extraction seed inputs are generated which give further fuzzing processes a *head start*. In our experiments, the lFUZZER-AFL combination achieves up to 17% more coverage on complex input formats like JSON, LISP, TINYC, and JAVASCRIPT compared to AFL.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Parsers;
- Theory of computation → Regular languages.

## KEYWORDS

fuzzing, test input generation, parser

### ACM Reference Format:

Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395363.3397348>

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397348>

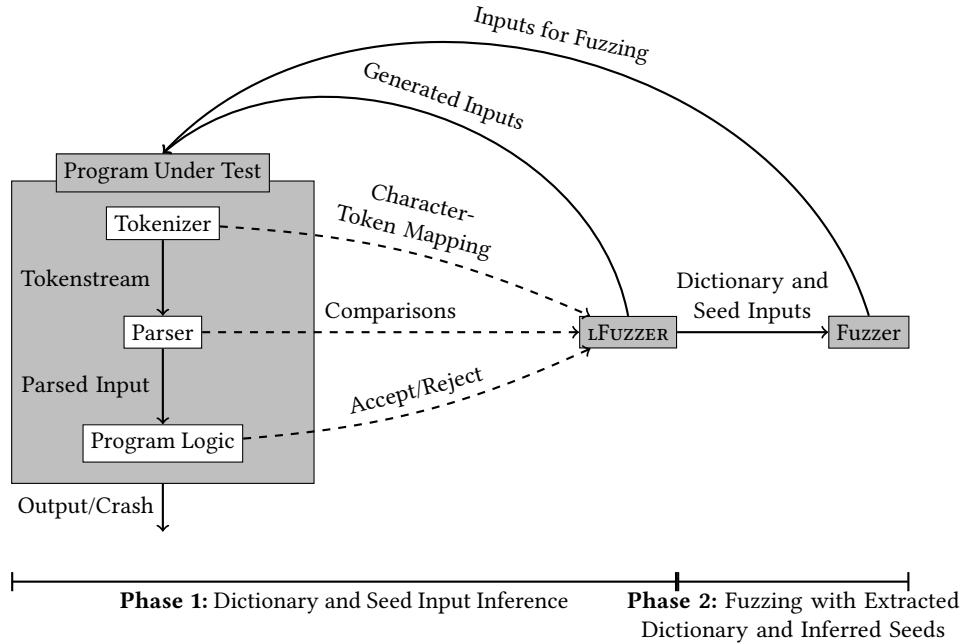
## 1 INTRODUCTION

Fuzzing has emerged as one of the key test-generation technologies in recent years. By automatically generating millions of tests in a short time a fuzzer is able to reveal bugs in software that may have stayed undetected by manually written tests. Fully automatic state-of-the-art fuzzers like AFL [19] explore the input space byte-wise and mostly randomly (with some coverage guidance). Thus, when fuzzing a program with a complex input structure, most of the generated inputs are invalid, and many features of the program cannot be covered as a random fuzzer is in general not able to produce keywords and complex structures: Generating a keyword like `while` randomly from letters has a chance of 1 in  $26^5$ . A pure random fuzzer will need a long time to generate this keyword, let alone the structures that follow it.

Several solutions have been introduced to circumvent this problem, the most important being *coverage guidance*. Maximizing code coverage, a random fuzzer like AFL over time is able to generate inputs consisting of different tokens, generating valid prefixes that cover more and more code until a valid input is finally composed. For constrained input languages, however, this is still not sufficient: As fuzzers compose their inputs character by character, they have to determine each and every keyword in the input again and again.

The problem of finding keywords and other lexical structures can be dramatically alleviated by providing a *dictionary* of common input fragments. This allows fuzzers to compose inputs from dictionary entries—in other words, they can build inputs from *tokens* rather than individual characters, which can be highly beneficial for fuzzing, e.g. with AFL. Even though AFL puts tokens from its dictionary randomly together, its coverage guidance can approximate the value of a generated input. This finally leads the random generation to inputs that survive the input validation stage and reach actual functionality—if a dictionary was supplied by the user.

In this paper, we propose a new approach that takes a program and *automatically* and *without seeds* extracts the tokens of its input language using dynamic tainting of implicit data transformations, to produce a dictionary and seed inputs to speed up fuzzing. The **key idea** of our approach, sketched in Fig. 1, is to systematically create inputs that cover all branches of a *tokenizer*—that is, a program part that composes characters into tokens. Our approach extends our earlier work [12], especially our tool pFUZZER. To this end, we dynamically track the *comparisons* made on input characters and tokens. The comparisons on input characters are easy to satisfy, as a tokenizer usually compares one or more input characters against a predefined set of keywords or special characters. It returns or stores a constant value (the token) based on those comparisons, which is again used in comparisons against other tokens in the parser. Hence, the dynamic tainting needs to be able to follow taints from



**Figure 1: How LFUZZER works.** In Phase 1, the learning phase, a dictionary and seed inputs are extracted. This is done as follows: LFUZZER generates an input and runs the program on this input. The tokenizer creates a token stream from the input (or the parser requests token after token from the tokenizer) and LFUZZER learns the mapping of each input character to the token it is converted to. The parser uses the tokens to parse the input, LFUZZER extracts the token comparisons made to generate the next input. If neither tokenizer nor parser reject the input, the program logic either outputs a result or crashes. In Phase 2, the *extracted dictionary* and *learned seed inputs* are used by a fuzzer to fuzz the program under test.

the input characters to the returned tokens to make use of the token comparisons in the parsing phase. Also, comparisons in the tokenizer that make up tokens can be used in fuzzing dictionaries.

In addition, we can use the token comparisons from the parser to create a *valid input*. We rely on the fact that a parser processes tokens one after another, comparing each input token against all valid tokens at this position before rejecting an input. We start with a random token which is likely rejected, extract the comparisons made on the token and replace it with one of the compared tokens, passing the first token comparison. We append new tokens until all token comparisons are passed. This input can be given as a seed to a fuzzer. Now we can create another seed input or start a fuzzer like AFL with the *extracted dictionary* and the *generated seed inputs*.

To the best of our knowledge, this is the first approach to *systematically taint, track and extract input tokens from a program under test* for the purpose of making test generation more efficient and more effective. Our token extraction approach solely relies on the comparisons made on input characters and tokens, making it easy to understand, implement, and extend for future research.

Our approach is effective. Our **evaluation** on six subjects ranging from csv to JAVASCRIPT shows that the dictionaries and seed inputs generated by our approach are more effective *and* more efficient for fuzzing. Compared to AFL and pFUZZER without any information, AFL with a dictionary of the string constants from the

program code and AFL with seed inputs generated by pFUZZER, AFL given our seeds and dictionaries achieves at least comparable, but in general higher coverage. As the benefits increase with growing complexity of the input language, our work opens the door towards highly efficient fuzzing of programs with complex input languages.

The remainder of the paper is organized as follows. Section 2 describes how we enable dynamic tainting of implicit data transformations and systematically explore the lexical input space of tokenizers, producing dictionaries and seeds for further fuzzing. Section 3 details the evaluation, comparing our approach against AFL and pFUZZER as described above. After discussing the related work (Section 4) on token extraction and dictionary usage, we discuss limitations and future work in Section 5, before concluding the paper in Section 6.

## 2 EXTRACTING INPUT TOKENS

Our goal is twofold: improving dynamic tainting by allowing implicit data conversions and using this extension for improved magic byte fuzz-blocking elimination.

First, we want to improve the precision of dynamic tainting by tracking taints on implicit data transformations. Such transformations are extensively used in input validators, more specifically in the tokenization phase of such a validator in which one or more input characters are converted to a constant value, a so called token.

Second, with this dynamic tainting extension we want to automatically generate both a dictionary and a set of seed inputs which can be used as a guidance for fuzzing using no more than the program as initial information. The key idea is to generate inputs that cover all branches of the *tokenizer*. Hence, we use a *test generator for tokenizers* to extract knowledge for another, general test generator.

Generating tests that cover tokenizers is a difficult task. On the one hand, random fuzzers like AFL fail in the presence of tokens, even if guided by coverage. On the other hand, testing approaches that solve path conditions to cover all branches are challenged by the complex path conditions in input processors [5]. Using a grammar or some other input model would dramatically increase the efficiency of fuzzing. Still, for many input formats no such model is available.<sup>1</sup> The technique of *parser-directed fuzzing* [12] aims to strike a compromise between the two, specifically generating inputs for *parsers* that process one element at a time. Thus, the key idea of our work is to

- (1) Extend dynamic tainting to implicit data transformations, using them to
- (2) Apply parser-directed fuzzing specifically to tokenizers, covering all branches and, consequently, all lexical elements (tokens) of the input language; and subsequently
- (3) Extract these tokens as dictionaries for effective fuzzing.

We implemented this approach in a tool called LFUZZER, being able to extract dictionaries and seed inputs from C programs.

## 2.1 Parser-Directed Fuzzing

As already mentioned, parsing makes an intensive use of implicit data conversions while also being hard to test with state-of-the-art fuzzers. Thus, we demonstrate the effectiveness of our approach by improving parser-directed fuzzing [12]. We extend our open-source implementation by Mathis et. al to also handle implicit data conversions. Hence, we shortly sketch the general idea of pFuzzer.

Parser-directed fuzzing [12] assumes that a parser processes inputs character by character, comparing each character against all expected characters at this position. Our earlier tool pFuzzer executes valid prefixes with random extensions and uses dynamic tainting to collect the comparisons done on the input characters. The collected comparisons are then used to find a valid substitution for the random extension.

We detail the approach on the example of an arithmetic expression parser: pFuzzer starts with a random character, e.g. '&', given to the program. This input is rejected but not before being checked if it is a *digit* or an *opening parenthesis*—the values a valid input for the expression parser can start with. In the next step pFuzzer replaces the random character with one of the values it was compared to, e.g. the character '1'. The input "1" is accepted, but pFuzzer can now decide to append another random character, looking for larger inputs. It could append '#' to the prefix "1", leading to "1#". '#' is compared against the characters that can follow a digit: '+' and '-' and replaced by one of them, leading to "1+" which is rejected. So pFuzzer appends another character, runs the program, analyzes

<sup>1</sup>Of course, if one writes a generic parser for a well known format (like JSON), a grammar would be available. Nonetheless, most of the time such formats are used to transport more specific data, e.g. for JSON a specific set of key-value pairs might be defined to exchange data between programs. With our approach it is possible to automatically extract such specific data without the need to define it manually.

---

### Algorithm 1 Token Taint Propagation Algorithm

---

```

1: lastTaint ← None
2: procedure PROPAGATE(Ins)
3:   if TYPE(Ins) = Comp ∧ IS_TAINTED(Ins) then
4:     lastTaint ← GET_TAINT(Inst)
5:   else
6:     if TYPE(Ins) ∈ {Assign, Return, Expr} ∧ HAS_CONSTANT(Ins) then
7:       ASSIGN_TOK_TAINT(Ins.getConstant, lastTaint)
8:     end if
9:   end if
10:  if TYPE(Ins) = Return then
11:    lastTaint ← None
12:  end if
13: end procedure

```

---

the comparisons made on this character (being again comparisons against *digits* and an *opening parenthesis*), replaces the appended character with one of them, resulting in the valid input "1+3".

In [12], we already mentioned tokenization as a strong limitation to fuzzing parsers and hence our approach. In the presence of a tokenization the character comparisons are shifted from the parser into the tokenizer. The tokenizer though compares every input character against *all characters and keywords* known to the program while a parser only compares against *valid characters* for the respective input position. In our example every input character would be compared against *digits*, '(', '+', '−', and ')'. Thus, pFuzzer may replace a randomly guessed character with an invalid value, e.g. the '&' pFuzzer started with in our example might be replaced with a ')'. Hence, the chances for pFuzzer replacing a guessed character with an incorrect value are higher, leading to more guesses until a valid input is composed. We believe that an explicit knowledge about a program's input tokens makes fuzzing much more efficient. Thus, we detail in the following how such tokens can be tainted, extracted and used for fuzzing (e.g. in parser-directed fuzzing).

We need to solve the following problems before we can use tokens in dictionaries and for seed input generation:

- (1) Linking of input characters to tokens (i.e. tainting the tokens with the taints of the originating characters).
- (2) Linking tokens to their actual meanings (e.g. a token T WHILE to the keyword while)<sup>2</sup>.

We will do this in three steps, sketching simple algorithms to explain our core ideas:

- (1) Detecting tokenization patterns and enable taint propagation to generated tokens;
- (2) Separating tokenizing and parsing code; and
- (3) Correcting misclassified taints on tokens.

Finally, we will also detail how tokens can be used to efficiently generate seed inputs for further fuzzing.

## 2.2 Propagating Token Taints

Adapting the dynamic tainting engine of pFuzzer means detecting and tainting tokens with the taints of the characters they are

<sup>2</sup>A necessary feature to use the token comparisons in the parsing phase for seed input generation equal to pFuzzer which relies on comparisons in the parsing phase.

derived from. Because one can imagine an infinite amount of possibilities to create a token from input characters, this detection can only be an approximation. Still, there are some general patterns that are used when tokenizing the input, which can be detected and handled by the method presented in Algorithm 1. Every tokenizer must compare one or more characters from the input explicitly against predefined values to be able to decide if a character belongs to a predefined token. After this comparison, the tokenizer should return or store a constant number – the token created from the characters. To avoid large overapproximations we restrict the "distance" between the comparison made and the generation of the token, i.e. the generation should happen in the function the character comparison is done or immediately after the function returns. Algorithm 1 implements this token detection approach.

The function PROPAGATE in Line 2 is called on every instruction that was executed during a run of the program under test. If a comparison with a tainted value is done (Line 3), we store the taint attached to the value (Line 4). The next time a constant is stored, returned, or used in an arithmetic expression (Line 6), the stored taint, flagged as a token taint, is attached to this constant (Line 7), as we assume this to be a token assignment. Such token taints are handled by the dynamic tainting engine as any other taint and are thus propagated like normal taints. If they appear in comparisons with other constants, a token comparison is reported and can then be used for generating seed inputs (cf. Section 2.5). Line 10 and Line 11 ensure that the taint stored in *lastTaint* is deleted on a function return, as tokenization rarely happens over returns.

In the following we present two different tokenization patterns and explain how they are handled by Algorithm 1:

**Basic.** The most basic conversion is a direct conversion, the input is compared against some expected character or a keyword and the respective token is assigned to some variable or returned:

```
void tokenize(char c) {
    if (c == '{')
        return L_BRACE;
}
```

In this case Algorithm 1 detects the comparison of *c* against '{' in Line 3, and will taint the constant *L\_BRACE* in Line 7 as it is a returned constant value which is detected by Line 6.

**Comparison Function.** Similarly to the basic conversion the comparison may be implemented in a custom function. Here we first taint the return value of the function and then taint the newly created token:

```
bool isLBrace(char c) {
    return c == '{';
}
int tokenize(char c) {
    if (isLBrace(c))
        return L_BRACE;
}
```

The comparison of *c* against '{' is done in the method *isLBrace*, the return value of the method is tainted by the standard dynamic tainting engine (*c* is tainted, thus the result of the comparison is tainted). In the function *tokenize*, this returned value is implicitly compared against the value *false* in the *if-condition*, triggering Line 3 and Line 4, storing the taint of the returned value for later usage

---

**Algorithm 2** Tokenization Phase Detection Algorithm

---

```
1: procedure DETECTTOKENIZATION(CallGraph)
2:   for node ∈ CallGraph do
3:     if HASCHARACTERCOMPARISON(node) then
4:       MARKTOKENIZE(node)
5:     end if
6:   end for
7:   while NEWNODEMARKED(CallGraph) do
8:     for node ∈ CallGraph do
9:       if ISPARENTTOKENIZE(node) then
10:        MARKTOKENIZE(node)
11:      end if
12:    end for
13:   end while
14: end procedure
```

---

in *lastTaint*. On the return of *L\_BRACE* the condition in Line 6 evaluates to true, hence the stored taint in *lastTaint* is used to taint the constant as in the basic case (Line 7).

### 2.3 Detecting the Tokenization Phase

The tainting engine may over-approximate and report wrong token comparisons (e.g. due to tainting a constant that is no token which is later used in comparisons). Algorithm 2 divides the program code in tokenizing and non-tokenizing based on the comparisons on input characters that are detected by the dynamic tainting engine. We designed this algorithm under the assumptions that parsing functions are never called by tokenizing functions and tokenization and parsing is strictly divided (so no function can be both at the same time). Thus, we can filter out token comparisons that are reported in the tokenizer, reducing the set of invalid token comparisons.

Algorithm 2 starts with the dynamic *CallGraph* of the application (Line 1) which is constructed during the program execution. The nodes (= functions) are iterated (Line 2) and functions containing input character comparisons (Line 3) are marked as tokenizing (Line 4). After that, the nodes are iterated (Line 8), and for each node it is checked if any of the parents (a caller of the function) is marked as tokenizing (Line 9). If so the respective node is marked as tokenizing as well (Line 10), finally approximating tokenizing and parsing code. The result is an underapproximation of the tokenizing code to avoid marking parser functions as tokenizing. This would hinder the input generation that uses the comparisons in the parser.

### 2.4 Correcting Misclassified Taints

It may happen that even after filtering token comparisons from tokenizing code, some reported token comparisons are still wrong token comparisons. This happens because of the over-approximation in the tainting engine marking any constant after a comparison as a token. Most of those constant values are only used within the tokenizing code and are therefore filtered by Algorithm 2, but some of them may also appear in the parsing code, leading to noise.

With Algorithm 3 we filter this noise by calculating which token values are used in majority on a specific input index<sup>3</sup>. The algorithm is based on the assumption that the noise, the wrongly reported

<sup>3</sup>Each character from the input has a fixed index that identifies the character. A taint also has the index information to map the taint back to the characters it stems from.

**Algorithm 3** Misclassified Taints Correction Algorithm

---

```

1: procedure CORRECTTOKENS(tokComps)
2:   majorityDict  $\leftarrow$  MAJORITYVOTE(tokComps)
3:   FILTERBYMAJORITY(tokComps, majorityDict)
4: end procedure
5:
6: procedure MAJORITYVOTE(tokComps)
7:   tokCounter  $\leftarrow$  DICTIONARY()
8:   for cmp  $\in$  tokComps do
9:     indexValue  $\leftarrow$  tokCounter.GET(cmp.idx)
10:    indexValue.INCREASE(cmp.val, 1)
11:   end for
12:   for el  $\in$  tokCounter do
13:     el.VALUE()  $\leftarrow$  MAX(el.VALUE())
14:   end for
15:   return tokCounter
16: end procedure

```

---

token comparisons, only appear in a small amount while the actual token comparisons form the majority. Thus, in the procedure MAJORITYVOTE (Line 6) we create a dictionary delivering for each index the most used token value. Concretely, we first iterate over all token comparisons (Line 8) and extract for each comparison the index information as well as the actual token value used (Line 9). Then, in Line 10 we count the number of token comparisons in which the token was tainted with the given index-value combination.

After all comparisons are analyzed, the algorithm evaluates the found numbers by iterating over all elements of the *tokCounter* dictionary (Line 12). Each element now contains a mapping from an index to the token values and their number of appearance during the execution. For example, we have the following comparisons:

```
(index: 5, tokenvalue: 10)
(index: 5, tokenvalue: 10)
(index: 5, tokenvalue: 1234)
(index: 5, tokenvalue: 10)
```

There are four comparisons on index 5, *three* with the token value 10 and *one* with the token value 1234—we assume the token value 10 to be correct. Hence, in Line 13 the index 5 is mapped to the token value 10 and then all token value mappings are returned in Line 15.

Finally, in Line 3 the majority dictionary from the MAJORITYVOTE function is used to filter all token comparisons on every index that have another token value than the one in the dictionary. In our example, the comparison with the value 1234 would be filtered.

## 2.5 Using Tokens for Seed Input Generation

Knowing the token definitions as early as possible is a crucial feature for successfully generating syntactically valid inputs. Hence, as soon as lFuzzer recognizes a new character or string in a comparison on an input character, it runs the program with the new value. For example, the first time lFuzzer finds a string comparison against *while*, it would run the program under test with the input *while* to extract the information to which token it is converted to. As the tokenizer is usually stateless, the keyword will be converted to a token and compared against all possible tokens a valid input can start with, giving us the wanted character-token mapping. With the knowledge about token comparisons and definitions we can track

**Table 1: The subjects used for the evaluation.**

Name	Accessed	Lines of Code
CSVPARSER	2018-10-25	297
INIH	2018-10-25	293
CJSON	2018-10-25	2,483
LISP	2019-03-19	2741
TINYC	2018-10-25	191
MJS	2018-06-21	10,920

the comparisons in the parsing phase and generate valid inputs as efficient as pFuzzer, *even if a tokenization phase is present*.

## 3 EVALUATION

### 3.1 Setup

We stick to the test subjects used in [12]<sup>4</sup>; details are listed in Table 1. The input languages range from simple formats as csv [9], INI [2], and JSON [4] up to complex formats like LISP [10], C (TINYC) [11], and JAVASCRIPT (MJS) [3]. As only two of the original subjects (MJS and TINYC) use tokenization<sup>5</sup> we decided to incorporate another complex subject with tokenization: a LISP interpreter.

For the evaluation we run lFuzzer in combination with AFL, i.e. we create a dictionary and a set of seed inputs with lFuzzer and then let AFL fuzz the subjects with this information. As baseline we compare against AFL<sup>6</sup>, run in two different modes. First, we run AFL with no dictionary and one test containing one whitespace as a seed input (since AFL requires a correct test to start fuzzing). Second, we run AFL given the set of strings extracted from the program under test, using the same seed as before. For getting those strings we first compiled the subject into a human readable bitcode format and then extracted the string literals by iterating over the global values of the bitcode file and writing the global strings to the AFL dictionary.<sup>7</sup> This delivers all string literals from the source code. Furthermore, to show that our changes made to pFuzzer in lFuzzer actually improves its fuzzing capabilities, we compare against pFuzzer in two different modes: first, pFuzzer is run until timeout, second, pFuzzer is used to extract seed inputs, then AFL is run until timeout with the extracted inputs. As pFuzzer does not extract any tokens, AFL is run without a dictionary.

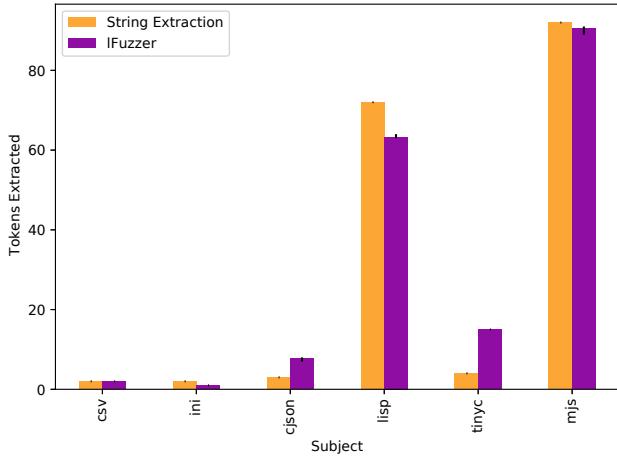
The evaluation was run for 24 hours per subject (excluding the static instrumentation and compilation time); the token learning and seed generation of lFuzzer is included in the 24 hours. The experiments were repeated *four* times to adhere to the non-determinism of all tools and were run on an Ubuntu 14.04.5 docker container with 3.3 GHz Intel processors, no tool was set up to use parallelization. Due to technical restrictions on our machine, AFL was run with AFL\_SKIP\_CPUFREQ enabled. We do **not provide**

<sup>4</sup>As in [12], the subjects are set up to read from the standard input (such that AFL can fuzz them), and to abort parsing on the first error with a non-zero exit code. MJS and LISP are changed such that semantic failures do not lead to a non-zero exit code.

<sup>5</sup>We still use the other subjects in the evaluation to show that lFuzzer does not harm the fuzzing process if no tokenization phase is available.

<sup>6</sup>As we assume nothing but the program as input to the fuzzer (no manual information like seed inputs or a dictionary), we only use fuzzers that meet this requirement.

<sup>7</sup>lFuzzer only requires LLVM bitcode to perform its analysis, so we decided to give AFL\_DICT the same level of abstraction.



**Figure 2: The number of valid tokens extracted per method and subject with error lines showing the minimal and maximal number of valid tokens extracted over all runs.**

any seed inputs to any fuzzer as we want to evaluate the input generation capabilities without additional knowledge.

To show the effectiveness of lFUZZER we evaluate the tools on three aspects:

**Token Extraction.** First, we show that on programs with complex input formats lFUZZER extracts tokens with higher precision and recall compared to naive string extraction.

**Code Coverage.** Second, we prove that our dictionary and seed generation improves coverage when fuzzing with AFL.

**Tokens Used.** Third, we look at how many tokens are actually used in the test inputs produced by any of the tools.

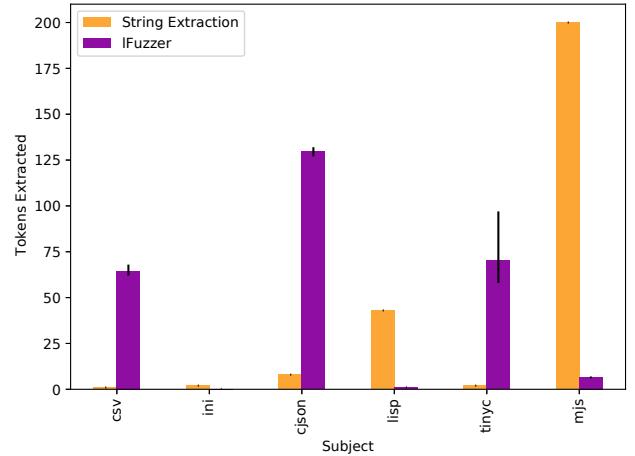
During our experiments we found out that the instrumented version of lISP produced by AFL has a bug resulting in segmentation faults. Thus, we had to exclude seed tests produced by lFUZZER and pFUZZER that start with “( # ” as otherwise AFL would not start.

### 3.2 Tokens Extracted

Fig. 2 shows how many valid tokens were found with the static string extraction and the active token learning of lFUZZER. As pFUZZER does not extract any tokens it will be omitted in this section. At first, one might think all tokens of a subject can be found with string extraction, but the results for cJSON and TINYC show another picture: lFUZZER finds more tokens than the string extraction. The missing keywords are single character tokens, e.g. a semicolon in TINYC as those are not present anymore in the bitcode. They are character constants in the original source code and as such compiled to integer constants in the bitcode.

For MJS and LISP, the picture changes: most of the existing tokens are present as strings in the code. lFUZZER misses some of them due to its dynamic token extraction—a token can only be found if it is seen during the learning phase.

When looking at the wrongly extracted tokens in Fig. 3, we first and foremost see that those subjects that do not have a tokenization phase tend to cause lFUZZER to report wrong tokens. We assume



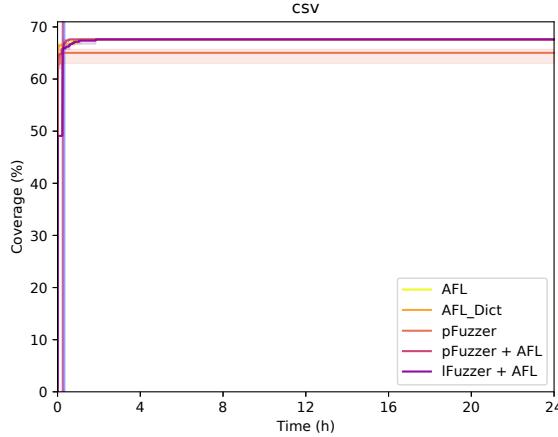
**Figure 3: The number of non-tokens (strings that are no tokens) extracted per method and subject with error lines showing the minimal and maximal number of non-tokens extracted over all runs.**

that some part of the code looks like a tokenization but accepts any character combination from the input, causing false positives. For TINYC, a lookahead causes a correctly detected token to be appended by other characters that followed the token while lFUZZER was running, resulting in some non-tokens being reported. Our token recognizer is designed to combine all characters to a token that were accessed between two token usages. In the case of TINYC more characters were accessed between two token usages than actually belonged to the token, hence this lookahead caused random characters to be appended to the extracted token values. This mostly happens for values with undefined length like variable identifiers or numbers as they need to be parsed character by character until an invalid character is detected (hence it is used in a comparison but not part of the token).

For MJS and LISP, the number of wrong tokens is very low compared to the string extraction method. The reason for this is twofold. First, programs with a more complex input format usually also contain a better error handling, trying to give the user a profound hint on why an erroneous input is actually invalid. Thus, different error messages have to be embedded in the code resulting in many different strings that are no valid tokens of the input language. Second, those subjects have a tokenization phase, hence lFUZZER is actually able to find and extract tokens and differentiate between actual tokenization code and the code that looks like such but is not.<sup>8</sup>

Table 2 shows the precision and recall of lFUZZER compared to naive string extraction on subjects with a tokenization phase (TINYC, LISP, MJS). The precision of lFUZZER is 27.7% higher compared to the precision of plain string extraction as our approach does not suffer from extracting strings that are not used as tokens but for example for error handling and user communication. Surprisingly though, the recall is 0.3% higher as well, coming from

<sup>8</sup>lFUZZER tries to find the tokenization part of the code which works well if there is a tokenizing code but may lead to false positives if no tokenizing code is present.



**Figure 4: Average, min- and maximum coverage for csv.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.

single character tokens that are compiled to integer constants in the LLVM bitcode and are thus not extracted.

*lFuzzer has a 27.7% higher precision and 0.3% higher recall regarding token extraction on subjects with a tokenization phase.*

### 3.3 Coverage

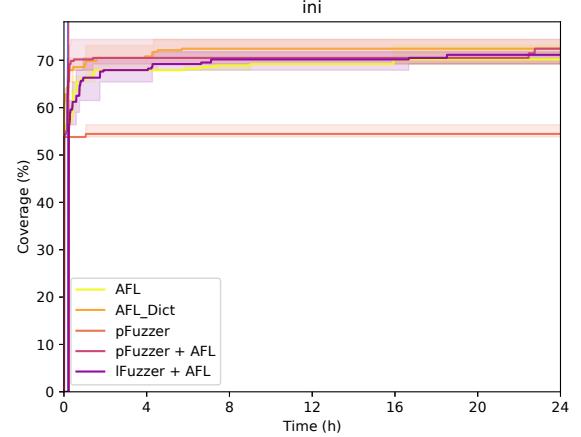
In the following we use branch coverage achieved by the syntactically valid inputs each tool generated.

**csv and ini.** In Fig. 4 and Fig. 5 we can see that programs with a simple input format can easily be covered by any tool. As all tools are based on random mutations and csv as well as ini do not have complex interdependent syntactic features, the whole feature space can easily be covered. For csv, a comma and a line break is sufficient to cover most of the code, ini's most complex input feature is a comment: arbitrary text surrounded by an opening and a closing bracket. pFuzzer misses some features and feature combinations leading to a lower coverage than AFL and lFuzzer can achieve. In combination with AFL the same coverage can be reached mainly because AFL on its own is able to achieve the coverage.

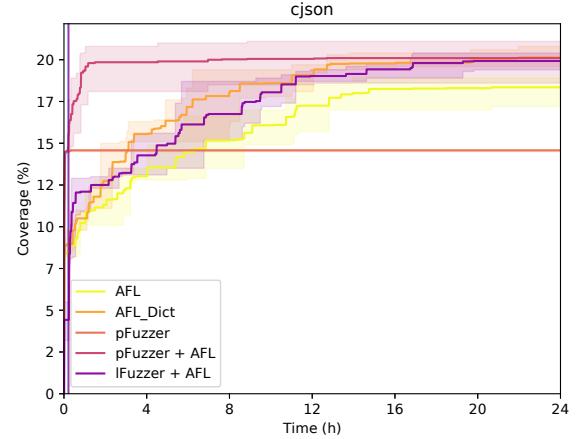
**JSON.** More interesting is the json subject which parses a much more complex input format, hence we see a slower and diverse increase in coverage over time for the different tools. The results in Fig. 6 show that almost all tools perform similarly good, with AFL\_DICT having the most coverage (20.2%, compared to pFuzzer

**Table 2: Precision and Recall on extracted strings regarding their token validity on subjects with a tokenization phase (TINYC, MJS, LISP).**

Tool	Precision	Recall
String Extraction	40.7%	88.5%
lFuzzer	68.4%	88.8%

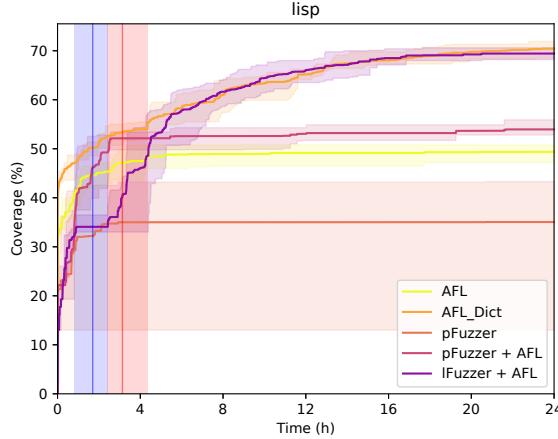


**Figure 5: Average, min- and maximum coverage for ini.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.

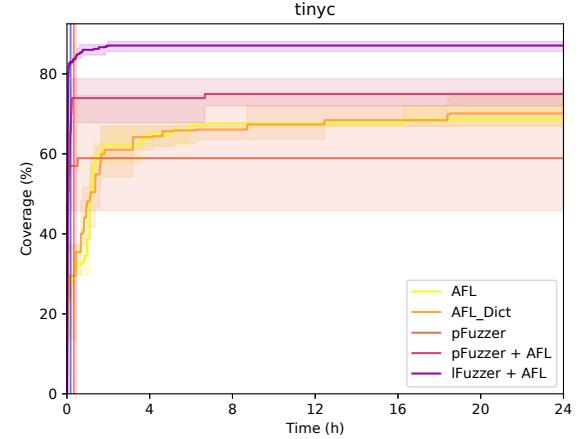


**Figure 6: Average, min- and maximum coverage for cJSON.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.

+ AFL having 20.1%, lFuzzer having 19.9%, AFL having 18.4%, and pFuzzer having 14.6%). The similar results can be explained as follows: AFL\_DICT has knowledge about the few existing keywords in cJSON, thus it is able to cover the code handling those, while also covering all the code AFL covers anyway. As cJSON does not have a tokenization phase, lFuzzer's token learning cannot come into play, resulting in falling back to using character comparisons. pFuzzer on the other hand is designed to work well on subjects with a parsing but no tokenization phase and thus covers some code very fast, in the long run though AFL is needed to cover code that pFuzzer cannot cover on its own. The low coverage all tools achieve is explained by the fact that cJSON contains generator code (code that creates a JSON string from a JSON object). In our experiments



**Figure 7: Average, min- and maximum coverage for LISP.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.



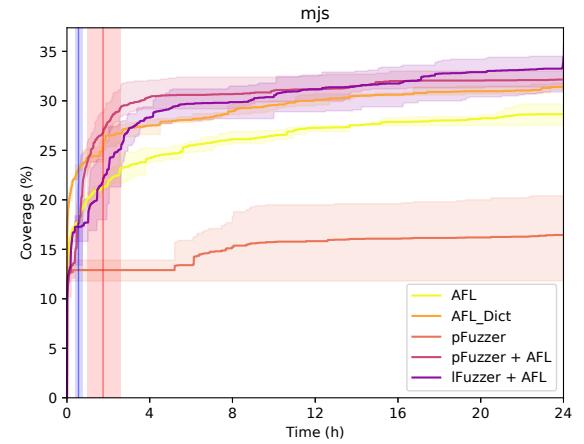
**Figure 8: Average, min- and maximum coverage for TINYC.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.

we focus on the parsing part of the program, hence the generator code is not triggered by the tools. As all tools cannot cover this part of the code, the comparison is still fair.

The missing tokenization phase also causes lFuzzer to incorrectly detect arbitrary character combinations as tokens which seemingly confuses AFL, resulting in a slower increase in coverage but ultimately leading to similar coverage as AFL\_Dict achieves.

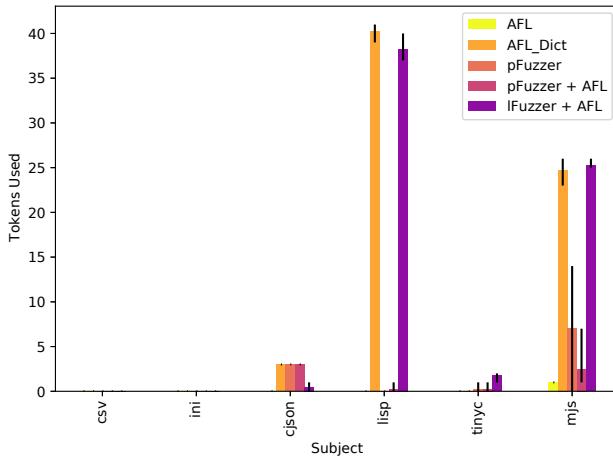
**LISP.** LISP, still having a simple syntax but a tokenization phase, shows how the generated dictionary and seed inputs of lFuzzer increase the performance of AFL. In Fig. 7 we can see that the lFuzzer-AFL combination achieves similar coverage as AFL\_Dict, having around 20% more coverage than AFL alone. For LISP, the seed inputs generated by lFuzzer are small and only cover a small part of the input space, still they are an efficient guidance for AFL to generate more complex inputs. LISP has a semantic phase which handles most of the keywords, “hiding” the actual token comparisons, making it impossible for lFuzzer to generate complex inputs. Many keywords are mapped to the same token and the token is then semantically analyzed, making it possible to extract the keywords but impossible to generate seed inputs (as the token comparisons used to generate those inputs are missing). This shows the great opportunities of our symbiotic approach—even if one of the tools alone fails to achieve coverage, the other tool is still able to address this flaw. Hence, pFuzzer alone is not able to produce a diverse set of inputs and thus achieves the worst coverage. AFL on the other hand profits from the seed inputs pFuzzer generates and covers more code compared to running alone.

**TINYC.** On TINYC on the other hand we can see the power of lFuzzer on its own. As shown in Fig. 8, the coverage our approach achieves is almost the maximum coverage reached throughout the fuzzing run, finally resulting in 17% more coverage than AFL\_Dict. lFuzzer successfully generates inputs that cover almost all the language features of TINYC, leaving out only a few keywords or feature combinations that are then filled by AFL shortly after it



**Figure 9: Average, min- and maximum coverage for MJS.** The red/blue vertical area indicates when lFuzzer/pFuzzer handed over to AFL including a solid line for the average.

started fuzzing. pFuzzer on the other hand struggles with the tokenization phase of TINYC (as mentioned in Section 2.1) resulting in a lower coverage than the one lFuzzer achieves. In combination with AFL the coverage is better than for AFL alone, but still worse compared to lFuzzer as the tokens are missing to support the AFL mutations. Those results can be explained by the structure of TINYC: every keyword has only a small feature space, i.e. each keyword is handled by a few lines of code. In contrast to that, MJS for instance has many internal functions, meaning that even if a successful call to the function is generated, the code coverage is highly dependent on the function arguments.



**Figure 10: The number of valid tokens with size greater than 3 used per tool and subject with minimum and maximum number of tokens used over several runs.**

**MJS.** On our most complex subject implementing an interpreter for a subset of JAVASCRIPT we can see the interplay of all components of lFUZZER in its full form. First, a precise set of tokens is learned and later given to AFL. Second, a diverse set of seed inputs is generated giving AFL a good starting point for further fuzzing. Hence, the lFUZZER-AFL combination outperforms AFL\_DICT, being faster in generating coverage and achieving 3% more coverage. pFuzzer again is blocked by the tokenization phase of MJS resulting in a low overall coverage. Together with AFL the coverage gets significantly better, still the tokens provided by lFUZZER improve the AFL fuzzing process even more. In contrast to cJSON, the low coverage of MJS results from the complex input format of this subject, making it hard for any approach to fully cover the subject.

*On subjects with complex input structures, lFUZZER achieves on weighted average 2.3% and up to 17% more coverage than AFL\_DICT.*

### 3.4 Tokens Used

Finally, in Fig. 10 we look at the tokens with more than three characters used in the syntactically valid inputs generated by each tool. Except for TINYC, AFL\_DICT uses a similar number of tokens. TINYC has only a small set of valid tokens, but those are used in complex structures, thus making it hard for a random approach like AFL to use them properly. On MJS and LISP AFL\_DICT and lFUZZER use almost the same number of tokens correctly in valid inputs, a result that is also reflected in the coverage graphs: on both subjects they achieve similar results. pFuzzer, also in combination with AFL, is not able to use a large set of diverse tokens in its generated inputs, being worse on almost all subjects, only for cJSON it is able to perform better than lFUZZER and equally well as AFL\_DICT. On MJS the pFuzzer-AFL combination uses fewer tokens than pFuzzer alone because AFL does not generate a large diverse set of tokens on its own and pFuzzer runs shorter compared to the solo run.

As we have already seen in Section 3.2, the input dictionary for both approaches has similarly many valid tokens. Even though the number of invalid tokens in the dictionary might deviate, AFL generates such a huge number of inputs in 24 hours that eventually the right tokens are used at the correct positions, achieving new coverage and thus guiding AFL towards a valid input.

*lFUZZER and AFL\_DICT are both able to use a large number of tokens (weighted average of 78% and 81% of all tokens) in valid inputs.*

In summary, the evaluation shows that (1) dictionaries are of great benefit for fuzzing, no matter if they are consisting of statically extracted strings or dynamically extracted tokens and (2) the combination of a precise dictionary and a set of diverse and valid seed inputs improves fuzzing on *languages with complex input formats* significantly, something only lFUZZER can achieve.

*In general, the more complex the input language, the greater the benefits of automatic dictionary extraction and seed input generation as done by lFUZZER.*

## 4 RELATED WORK

To the best of our knowledge, we are the first to dynamically taint, track and extract tokens from a program under test for more efficient fuzzing. Therefore, we will look into the research done on dictionaries and their optimization and usage for fuzzing.

### 4.1 pFuzzer

First and foremost, we have to mention our own work pFuzzer [12], as this work builds on the approach and research results. In pFuzzer, we were targeting parsers and generate inputs that survive the parsing stage and are able to test the program logic. This is done by tracking the comparisons done on the input characters and using them to systematically build a valid input. Each time a character is rejected it gets replaced by one of the values it was compared to, iteratively creating an input that survives more and more comparisons in the parser until it is finally accepted by the parser. We extended the pFuzzer work by improving the dynamic tainting technique to not only track character comparisons but also taint tokens that are implicitly composed from input characters; enabling the usage of token comparisons<sup>9</sup>. Using the extended dynamic tainting technique, lFUZZER automatically identifies and extracts tokens from the source code, using them (1) for generating a dictionary that can be used for further fuzzing and (2) for generating seed inputs by adapting their iterative input creation technique but lifting it to the token comparison level.

### 4.2 Learning Input Structure

Maybe the closest approach to ours is the one by Shastry et al. [16], statically inferring a dictionary from the source code. They apply backward slicing and control-flow graph analysis to find tokens and token conjunctions that can be used in a dictionary to improve fuzzing. In contrast to their approach we are using dynamic program analysis which is more robust to unusual code patterns. While

<sup>9</sup>In [12], we explicitly mentioned token conversions as a limitation, saying that this prevents pFuzzer from testing complex input processors efficiently.

Shastry et al. need heuristics to find the comparisons and values for their analysis, we can simply observe all comparisons in the program with dynamic tainting and report the comparison values. Static code analysis may miss dynamic code features (like an array which is filled with keywords in a loop), which may lead to omitted tokens. A keyword detection algorithm might be implemented as follows:

```
// tokenorder: T_WHILE, T_IF, T_UNDEF
char* kwds[] = {"while", "if", null};
bool isKeyword(char* c) {
    int sym = 0;
    while (kwds[sym] != 0
        && strcmp(c, kwds[sym]) != 0) {
        sym++;
    }
    return sym;
}
```

The keywords array could also be filled dynamically. For a static approach it is very hard to extract the keywords from the array. For a dynamic approach, no matter how the array is initialized, the `strcmp()` function will be called with all values in the array (assuming no comparison matches), thus making it easy to extract the keywords for a dictionary construction. Furthermore, *our approach also provides a set of seed inputs that is used to give the fuzzer a head start for fuzzing* by providing a set of valid syntactic structures of the input format that can be used for recombination and mutation.

Different approaches have been used to learn inputs or their structure and improve fuzzing with the gained knowledge. Höschele et al. [8] presented an approach to learn the input format in form of a context-free grammar from the program under test. They use a set of valid seed inputs to explore the program execution and infer a grammar based on the program structure and the consumption of input characters during parsing. Similar to our approach they are using the structural information of a program to gain input format knowledge; but first, *we are not relying on seed inputs, we generate them* and second, we do not extract the full input format but only the tokens used by the program. A combination of both approaches might be beneficial though, as detailed in Section 5.3.

Godefroid et al. [6] apply recurrent neural networks to learn the statistical distribution of input elements from a large corpus of valid inputs and then generate new inputs with the neural network. In contrast to our approach, they do not extract the input elements explicitly but encode the knowledge in a neural network, making it not accessible out of the box for existing fuzzing techniques. Furthermore, the corpus of inputs to learn from needs to be large to train the neural network while *our approach extracts tokens and seed inputs with having nothing more than the program under test*.

### 4.3 Selecting and Using Seeds

With REDQUEEN, Aschermann et al. [1] presented an approach which made it possible to circumvent different fuzzing roadblocks, among others magic bytes. In general they rely on a similar approach as the authors of VUZZER [14], observing the control and data flow of an application and finding parts of the input that belong to branching conditions. Hence, these tools which rely on dynamic tainting can make use of our improved tainting framework to also

observe token comparisons. We lift the token comparisons back to the character comparisons they represent enabling the magic byte solving to also work beyond the tokenizer. With a feedback loop portions of the input are gradually replaced with different characters until the branching condition switches and a new branch is taken. This is similar to our seed generation technique, but we are explicitly tracking the data flow, even beyond tokenization, and are replacing rejected input values with the values they were compared to, even if the comparison was done on the token level. Our approach systematically constructs diverse inputs that survive the parsing stage and test the program logic.

Several works focus on the problem of *seed input selection*. Wang et al. [18] generate seeds via analyzing the corpus to learn a probabilistic context-sensitive grammar and using this grammar together with mutations to create a set of seeds that cover the least used features from the original seed set. Rebert et al. [15] looked at different seed selection strategies and evaluated them on different subjects to find out how seed selection influences the result of the fuzzing session. They found out that seed selection can actually help improving the fuzzing performance. In any case, seed selection assumes seeds to be present to select from. In contrast, our approach creates a set of seed files, not needing any starting input. Still, the seeds we currently generate are not optimized in any way, hence seed selection might further increase the fuzzing performance.

## 5 LIMITATIONS AND FUTURE WORK

As shown in the evaluation, our approach is able to taint and track tokens during program execution which improves fuzzing significantly on subjects with complex input structures. Still, some limitations remain to be solved in future work; we list them in the following and provide ideas for solutions:

### 5.1 Parsing Style

Similar to [12], our technique is limited to recursive-descent parsers, relying on some assumptions: First, the program under test needs to have a tokenization phase, meaning that there is a part in the code which consecutively compares slices of the input against predefined characters and keywords and forwards the resulting tokens one after another to the parser. As this is the textbook approach to writing input processors for complex input formats, we believe that most of the handwritten parsers are designed like this. Second, we rely on dynamic tainting to track the input characters and the comparisons made on them throughout the program execution.

Other parsing styles like table-driven parsing, the common alternative to recursive-descent parsing, have a fundamentally different structure which we are not able to analyze at the moment. It might be possible to adapt the techniques presented in this paper to other parsing styles. Still, it is questionable if this is beneficial for the following reasons: First, 80% of the top 17 programming languages on Github are recursive-descent parsers [12] (with CLANG [17] and GCC [13] being the most famous ones), hence only 20% are implemented differently. Second, table driven parsers are usually not manually written but generated from a machine readable grammar. Hence, one can apply *grammar-based fuzzing* which will be superior to character-based fuzzing.

## 5.2 Detecting Patterns

C is a language that is known for its freedom. A programmer can solve a problem in many different ways, some of them might be considered as straight-forward while others might be more specialized. The creation of a tokenizer is not excluded from this. While we are confident that we covered the typical tokenization patterns it might happen that we miss the generation of specific tokens or even the generation of all tokens. Our evaluation has shown that most input parsers are implemented close to the textbook approach, making it possible for us to extract the tokens we want.

## 5.3 Extracting Input Models

Tokens are not only valuable to our previous work pFuzzer [12], they can also be beneficial when combined with *grammar learning techniques*. AUTOGRAM [8] and MIMID [7] implement approaches to infer context-free grammars from a program under test, tracking how individual input characters are processed within the program; AUTOGRAM focuses on data flows, whereas MIMID uses control flow instead. The extracted tokens might serve as a great base for both approaches to construct terminals in the grammar. We would lift the minimal building blocks for the grammar from single characters to full tokens, making it easier to construct a grammar.

## 6 CONCLUSION

Fuzzing is one of the key technologies for software testing, currently experiencing a renaissance in research and industry. Improved methods made it possible to automatically test a wide variety of programs. Testing the actual functionality of programs with complex input formats though is a challenge that remains to be solved until today; state-of-the-art fuzzers mostly test the input rejection capabilities of the software under test rather than the actual functionality. With our implicit-data-transformation tainting, dictionary extraction, and seed input generation methods we make it possible to help fuzzers go beyond the input validation stage and test the actual program functionality.

Our approach is based on the observation that tokenizers are in general implemented by the book: one or more input characters are compared against predefined values and if one value matches the respective token is forwarded to the parser. We can find those patterns in the program execution using them for an *enhanced taint tracking*, extract the values to *generate a dictionary*, and also *build valid and diverse seed inputs* token by token. Thus, we are able to produce a foundation for fuzzing, outperforming AFL without any information and AFL given the strings from the program as dictionary while still being fully automatic, using nothing more than the source code of the program.

Even though our results are very promising, this approach just serves as a foundation to show the potential of token extraction for fuzzing. Future combinations with more sophisticated techniques like grammar learning and following grammar-based fuzzing may result in even more efficient and effective testing. With this work we want to set one more milestone on the road towards efficient, effective, and fully automatic fuzzing of programs with complex input structures.

We are determined to making our research public and reproducible. lFuzzer and all evaluation data is available as open source at the project page:

<https://github.com/uds-se/lFuzzer>

## ACKNOWLEDGMENTS

This work was partially supported with funds from the Bosch Research Foundation in the Stifterverband (Reference: T113/33825/19). We thank Andreas Zeller's team at CISPA and the anonymous reviewers for their great feedback.

## REFERENCES

- [1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019*. <https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/>
- [2] Ben Hoyt and contributors. 2018. inih - Simple .INI file parser in C, good for embedded systems. <https://github.com/benhoyt/inih>. Accessed: 2018-10-25.
- [3] Cesanta Software. 2018. Embedded JavaScript engine for C/C++. <https://mongoose-os.com>. <https://github.com/cesanta/mjs>. Accessed: 2018-06-21.
- [4] Dave Gamble and contributors. 2018. cJSON - Ultralightweight JSON parser in ANSI C. <https://github.com/DaveGamble/cJSON>. Accessed: 2018-10-25.
- [5] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (Tucson, AZ, USA) (ACM SIGPLAN Conference on Programming Language Design and Implementation)*. ACM, New York, NY, USA, 206–215.
- [6] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *IEEE/ACM Automated Software Engineering*. IEEE Press, 50–59.
- [7] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*.
- [8] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *IEEE/ACM Automated Software Engineering (Singapore, Singapore) (ASE 2016)*. ACM, New York, NY, USA, 720–725. <https://doi.org/10.1145/290276.2970321>
- [9] JamesRamm and contributors. 2018. csv\_parser - C library for parsing CSV files. [https://github.com/JamesRamm/csv\\_parser](https://github.com/JamesRamm/csv_parser). Accessed: 2018-10-25.
- [10] Justin Meiners. 2019. Embeddable lisp interpreter written in C. <https://github.com/justinmeiners/lisp-interpreter>. Accessed: 2019-03-19.
- [11] Kartik Talwar. 2018. Tiny-C Compiler. <https://gist.github.com/KartikTalwar/3095780>. Accessed: 2018-10-25.
- [12] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (Phoenix, AZ, USA) (PLDI 2019)*. ACM, New York, NY, USA, 548–560. <https://doi.org/10.1145/3314221.3314651>
- [13] Joseph Myers. 2008. New\_C\_Parser. [http://gcc.gnu.org/wiki/New\\_C\\_Parser](http://gcc.gnu.org/wiki/New_C_Parser). Accessed: 2019-05-09.
- [14] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium*.
- [15] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Conference on Security Symposium (San Diego, CA) (SEC'14)*. USENIX Association, Berkeley, CA, USA, 861–875. <http://dl.acm.org/citation.cfm?id=2671225.2671280>
- [16] Bhargava Shastray, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 26–47.
- [17] The Clang Team. 2019. Clang - Features and Goals. <http://clang.llvm.org/features.html#unifiedparser>. Accessed: 2019-05-09.
- [18] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy*. IEEE, 579–594.
- [19] Michal Zalewski. 2015. afl-fuzz: making up grammar with a dictionary in hand. <https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html>. Accessed: 2019-05-07.

# Fast Bit-Vector Satisfiability

Peisen Yao

The Hong Kong University of Science and Technology,  
China  
pyao@connect.ust.hk

Heqing Huang

The Hong Kong University of Science and Technology,  
China  
hhuangaz@cse.ust.hk

Qingkai Shi

The Hong Kong University of Science and Technology,  
China  
qshiaa@cse.ust.hk

Charles Zhang

The Hong Kong University of Science and Technology,  
China  
charlesz@cse.ust.hk

## ABSTRACT

SMT solving is often a major source of cost in a broad range of techniques such as the symbolic program analysis. Thus, speeding up SMT solving is still an urgent requirement. A dominant approach, which is known as the eager SMT solving, is to reduce a first-order formula to a pure Boolean formula, which is handed to an expensive SAT solver to determine the satisfiability. We observe that the SAT solver can utilize the knowledge in the first-order formula to boost its solving efficiency. Unfortunately, despite much progress, it is still not clear how to make use of the knowledge in an eager SMT solver. This paper addresses the problem by introducing a new and fast method, which utilizes the interval and data-dependence information learned from the first-order formulas.

We have implemented the approach as a tool called TRIDENT and evaluated it on three symbolic analyzers (ANGR, QSYM, and PINPOINT). The experimental results, based on seven million SMT solving instances generated for thirty real-world software systems, show that TRIDENT significantly reduces the total solving time from 2.9 $\times$  to 7.9 $\times$  over three state-of-the-art SMT solvers (Z3, CVC4, and BOOLECTOR), without sacrificing the number of solved instances. We also demonstrate that TRIDENT achieves the end-to-end speedups for three program analysis clients by 1.9 $\times$ , 1.6 $\times$ , and 2.4 $\times$ , respectively.

## CCS CONCEPTS

• Theory of computation → Automated reasoning; Program analysis.

## KEYWORDS

Satisfiability modulo theory, SAT solving, program analysis

### ACM Reference Format:

Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast Bit-Vector Satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397378>

Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397378>

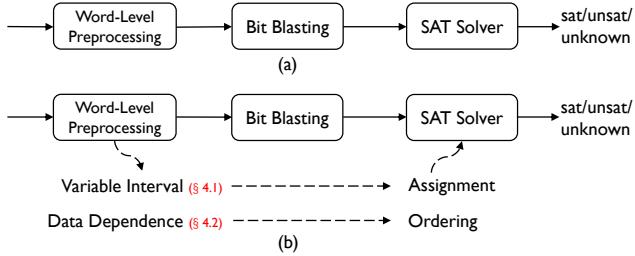
## 1 INTRODUCTION

Satisfiability Modulo Theories (SMT) solving has undergone steep development during the last decade, enabling a wide range of practical applications, such as test case generation [20, 40], static bug finding [77, 86], and program repair [62]. SMT solvers decide the satisfiability of formulas over first-order theories. Among the many theories within the SMT-LIB initiative [3], the theory of bit-vector is of crucial importance for software analysis, due to its capability of faithfully and precisely modeling the bit-level behavior of the machine instructions [86].

Existing SMT solving algorithms can be classified as being lazy or eager [15, 51]. The latter is the predominant approach to solve bit-vector constraints [43]. As illustrated in Figure 1(a), an eager bit-vector solver first simplifies the input formula with the word-level simplification rules [1, 17], and then translates a bit-vector formula into an equisatisfiable Boolean formula via bit-blasting, which is finally solved by an SAT solver. In practice, the word-level simplification and bit-blasting phases are efficient. However, the SAT solving phase is often the performance bottleneck due to its NP-completeness [43, 50].

While a significant effort has been investigated in the lazy SMT solving [9, 22, 42, 75], to the best of our knowledge, there is little progress on improving the SAT solving algorithms for an eager bit-vector solver. Most existing effort either focuses on enhancing the word-level preprocessing phase via different simplifications and semi-decision procedures [34, 36, 44, 61], or aims to devise better encoders for bit-blasting [45, 57]. These methods attempt to reduce the overhead of SAT solving by translating the bit-vector formulas into smaller Boolean formulas. However, the SAT solver itself still uses the default strategy irrespective of the word-level problem characteristics, thus losing many optimization opportunities.

In this paper, we present an approach that significantly improves the bit-vector SMT solver by leveraging the word-level information, just as shown in Figure 1(b). Our key insight is that the word-level information inferred *before* bit-blasting can be preserved to boost the SAT solving phase. Specifically, a conventional SAT solver often works as follows: given a Boolean formula, the SAT solver checks its satisfiability by choosing a variable in the formula, assigning a truth value to it, simplifying the formula based on the assignment, and then recursively checking if the simplified formula is satisfiable.



**Figure 1: (a) Conventional workflow of eager SMT solving. (b) Our workflow of eager SMT solving.**

If the search fails, the same recursive check is performed assuming the opposite truth value. Clearly, the performance of the SAT solver crucially depends on the *branching heuristic* [12, 47, 54, 59, 85], which concerns both the order in which Boolean variables are chosen for assignment and the values assigned to them. Therefore, we can utilize two categories of word-level information — data dependence and variable interval — to guide the decision order and the assignments, respectively.

We have implemented our approach as a tool called TRIDENT on top of the Z3 SMT solver [29]. We evaluate TRIDENT on three realistic symbolic analysis platforms (ANGR [78], QSYM [88], and PINPOINT [77]), which generate a large set of, nearly 7.4 million, bit-vector constraints from thirty real-world software systems. The experimental results show that, compared to Z3, TRIDENT achieves 3.4 $\times$  to 6.3 $\times$  speedup (with 4.9 $\times$  on average). TRIDENT also outperforms two other state-of-the-art SMT solvers, CVC4 and BOOLECTOR, achieving 7.9 $\times$  and 2.9 $\times$  speedup, respectively. Armed with our new bit-vector solver, the three symbolic analysis platforms, ANGR, QSYM, and PINPOINT, achieves 1.9 $\times$ , 1.6 $\times$ , and 2.4 $\times$  speedups, respectively. To sum up, we make the following main contributions in the paper:

- We introduce an approach to scaling eager bit-vector solvers, which leverages interval and data-dependence information to guide the branching heuristic in the SAT solving phase.
- We implement the proposed approach and apply our solver to three symbolic analysis applications, including control-flow recovery, test case generation, and static bug hunting.
- We conduct a thorough evaluation, confirming the effectiveness of our approach. We also provide a new set of publicly-available benchmarks with nearly seven million SMT solving instances, which can help better evaluate SMT solvers.

## 2 PRELIMINARIES

This section introduces the basic concepts and terminologies used in the paper.

### 2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some first-order theories. Examples include the theories of linear arithmetics, bit-vectors, arrays, lists, and strings.

A bit-vector is a fixed sequence of bits. The theory of quantifier-free bit-vectors (QF\_BV) is a many-sorted first-order theory. The length of a bit-vector is usually referred to as bit-width, and bit-vectors with different widths correspond to different sorts. The functions in the theory include  $+$ ,  $-$ ,  $\times$ ,  $\div$ ,  $\&$ ,  $|$ ,  $\oplus$ ,  $\ll$ ,  $\gg$ , *concat*, and *extract*, interpreted as addition, minus, multiplication, division, bit-wise and, bit-wise or, bit-wise exclusive or, left-shift, right-shift, concatenation, and extraction, respectively. The predicates include  $=$ ,  $<$ , and  $\leq$ , which are interpreted as “equal to”, “less than”, and “less than or equal to”, respectively. For the ease of presentation, we assume that all bit-vector variables represent unsigned integers. Deciding the satisfiability of a quantifier-free bit-vector formula is NP-complete, or more exactly NEXPTIME-complete [50].

### 2.2 Eager Bit-Vector Solving

Existing SMT solving algorithms can be classified as being *lazy* or being *eager*. Given a first-order formula, a lazy approach [5, 17, 75] usually iteratively refines an over-abstraction of the formula by combining a SAT solver and a theory solver. First, it abstracts the first-order formula by replacing each atomic predicate<sup>1</sup> with a distinct Boolean variable, producing a formula called the *Boolean skeleton*. Then, it uses the SAT solver to iteratively enumerate models of the Boolean skeleton, and the theory solver to check these models for satisfiability. In contrast, an eager approach [36, 48] translates the first-order formula, in a single satisfiability-preserving step, into an equisatisfiable Boolean formula, which is delegated to a SAT solver for determining the satisfiability, as shown in Figure 1.

*Example 2.1.* Consider the bit-vector formula  $\phi \equiv x \geq 3 \wedge (x = 0 \vee y = 4)$ , where  $x$  and  $y$  are two bit vectors. A lazy SMT solver will abstract  $\phi$  as a Boolean skeleton  $b_1 \wedge (b_2 \vee b_3)$ , where the Boolean variables  $b_1, b_2$ , and  $b_3$  represent  $x \geq 3, x = 0$ , and  $y = 4$ , respectively. Suppose that its SAT solver first suggests a model ( $b_1 = 1, b_2 = 1, b_3 = 0$ ). The Boolean model is mapped back as conjunctions of atomic predicates  $x \geq 3 \wedge x = 0 \wedge y \neq 4$ , which is then checked by the theory solver. If the conjunctions are satisfiable, so is the original bit-vector formula. Clearly, the conjunctions are unsatisfiable, meaning that the Boolean model is spurious. Then, its SAT solver attempts to suggest another model of the Boolean skeleton. In the worst case, the SAT solver enumerates all models of the Boolean skeleton, which is expensive. In contrast, the eager approach translates  $\phi$  into an equisatisfiable Boolean formula and only calls a SAT solver once to determine the satisfiability.

In the eager approach to SMT solving, the typical method for translating a bit-vector formula into an equisatisfiable Boolean formula is *bit-blasting*. The procedure encodes a bit-vector variable using a sequence of auxiliary Boolean variables, each of which represents a bit of the bit-vector variable. Bit-vector functions such as addition and multiplication are modeled using Boolean connectives in a way that mimics the hardware circuits of these functions [15, 51]. Note that in order to encode these functions more compactly, the bit-blaster also introduces a sequence of auxiliary Boolean variables to represent each *bit-vector term*, i.e., functions

<sup>1</sup>An atomic predicate is a Boolean-type expression without Boolean connectives such as  $x + y = 3$ .

**Algorithm 1:** Eager bit-vector solving

---

```

Input: A bit-vector formula  $\phi$ .
Output: satisfiable or unsatisfiable.
1  $\phi' \leftarrow$  apply word-level simplifications to  $\phi$ ;
2  $BF(\phi') \leftarrow$  apply bit-blasting to  $\phi'$ ;
3  $BV(\cdot) \leftarrow$  the set of Boolean variables in  $BF(\phi')$ ;
4 return CDCL_SAT( $BF(\phi')$ );
5 Function CDCL_SAT( $\phi$ ):
6   while true do
7     if there is an unassigned Boolean variable then
8       select and assign a variable from  $BV(\cdot)$ ;
9     while BCP() == CONFLICT do
10       $(C, level) =$  conflict_analysis();
11      if level < 0 then
12        return unsatisfiable;
13      else
14        backtrack( $level$ );
15        add_clause( $C$ );
16    else
17      return satisfiable;

```

---

applied to a set of bit-vector variables.<sup>2</sup> In the remainder of the paper, we denote by  $BF(\phi)$  the Boolean formula encoding a bit-vector formula  $\phi$ . We denote a  $n$ -bits bit-vector variable or term  $t$  as  $t = [b_{n-1}, b_{n-2}, \dots, b_0]$ , where  $BV(t) = \{b_{n-1}, b_{n-2}, \dots, b_0\}$  is the set of Boolean variables encoding  $t$ . Since we assume that bit-vectors are unsigned, we have

$$t = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_0 \times 2^0, \quad (1)$$

which means that once the values of Boolean variables  $BV(t)$  are fixed, we can obtain the value of  $t$ , and vice versa.

*Example 2.2.* Let  $x$  and  $y$  be two 8-bits bit-vectors such that  $x = [a_7, \dots, a_0]$  and  $y = [b_7, \dots, b_0]$ . Consider a term  $x \mid y$ , which is the “bit-wise or” of the variables  $x$  and  $y$ . To encode the term, the bit-blaster first introduces eight Boolean variables,  $c_0, c_1, \dots, c_7$ , to represent the calculation result, i.e.,  $x \mid y = [c_7, \dots, c_0]$ , and then translates the term into the following Boolean formula that encodes the semantic of “bit-wise or”:

$$\bigwedge_{i=0}^7 ((a_i \vee b_i) \leftrightarrow c_i)$$

One purpose for introducing the auxiliary variables  $c_i$  is that we can reuse the translation of the term  $x \mid y$  when the term is used many times in a constraint like  $x \mid y + z > 10 \wedge x \mid y - z < 5$ .

### 2.3 SAT Solving and Branching Heuristic

To determine the satisfiability of a Boolean formula generated by bit blasting, a Conflict-Driven Clause-Learning (CDCL) SAT solver will be employed [11, 80]. The function CDCL\_SAT of Algorithm 1 presents the basic CDCL search loop. At each step, the branching heuristic picks an unassigned variable and assigns it a truth value

<sup>2</sup>More exactly, a term is inductively defined as either a variable or a function applied to any number of other terms. We differentiate between variable and term for the sake of presentation.

of 1 or 0 (Line 8). The picked variable is called the decision variable. Then, the solver uses a method called Boolean Constraint Propagation (BCP) (Line 9) to simplify the formula, by leveraging the current assignment and its logical consequences. If the propagation leads to a falsified clause, a conflict occurs, indicating that a previous decision is not appropriate. The level<sup>3</sup> of that decision is identified by the conflict analysis (Line 10), following which the solver recovers from the conflict by backtracking, undoing the offending decision, and trying some other assignments. A clause learned from the conflict is also added to the original formula (Line 15), to prevent the search from repeating the mistake. The loop repeats until all clauses are satisfied, or some conflict cannot be resolved by backtracking and, thus, the formula is unsatisfiable (Line 12).

*Example 2.3.* Consider the Boolean formula  $(a \vee b) \wedge (\neg a \vee b)$ . If we pick the Boolean variable  $a$  as the decision variable and assign 1 to it, the BCP can infer than  $b$  should also be 1, because otherwise the clause  $(\neg a \vee b)$  would be falsified.

*Example 2.4.* Consider the Boolean formula  $(a) \wedge (\neg a \vee \neg b) \wedge (b \vee c)$ . Suppose that we first choose  $a$  and assign 1 to it. The BCP can infer that  $b$  should be 0 and further deduce that  $c$  should be 1. All clauses are satisfied now. In total, only one decision is made. However, if we first choose  $c$  and assign 0 to it, the BCP can infer that  $b$  should be 1 and further infer that  $a$  should be 0. The current assignment ( $c = 0, b = 1, a = 0$ ) results in a falsified clause  $(a)$ . The SAT solver has to perform backtracking, undoing the first decision and continuing a new round of search.

CDCL SAT solvers crucially depend on the branching heuristic for their performance. A state-of-the-art branching heuristics Variable State Independent Decaying Sum (VSIDS) [59] maintains a score for each Boolean variable throughout the search. It initializes the score based on the globally statistical information, such as the number of clauses in which a variable appears. At each step, VSIDS selects the variable with the highest score as the next decision variable. The scores are adjusted periodically by aggregating a variable’s effects in the previous conflicts.

## 3 MOTIVATION

In this section, we use several examples to motivate our approach. We show that the bit-vector level information can be leveraged to partially determine the assignments (§ 3.1) and the order of decision variables (§ 3.2).

### 3.1 Assignment Restriction

In program analysis, to achieve the bit-level precision, an integer variable in a program is usually modeled as a bit-vector, of which the length is the bit width of the variable’s type. The constraints in a program analysis consist of the operations and the relations among the bit-vector variables. In theory, a 32-bit unsigned bit-vector variable represents a large range of values, i.e., from 0 to  $2^{32} - 1$ . Besides, the number of variables in a path constraint can be huge. As the consequence, such a vast search space stresses the capability of the constraint solver, causing significant performance issues.

<sup>3</sup>The decision level of a variable is the number of decision variables occurring before the variable.

Fortunately, in practice, we find that the feasible solution space of the variables can be small. More specifically, we observe that, in real-world programs, not all statements are complex, e.g., containing non-linear computation and library function calls. Most variables are only involved in simple statements, such as linear assignments and linear branch conditions. By inspecting the constraints encoded for these simple statements, we can soundly approximate the solution space of the bit-vector variables and, thus, reduce the search space of SAT solving.

*Example 3.1.* Suppose that  $x = [a_7, \dots, a_0]$  and  $y = [b_7, \dots, b_0]$  encode two 8-bits unsigned integer variables and the formula  $\phi \equiv y = x + 1 \wedge y < 5 \wedge x * x < 60$  is a constraint generated by a program analyzer. After bit-blasting the formula, we obtain a Boolean formula  $BF(\phi)$ . If we are able to infer that  $x < 4$  from the linear constraint  $y = x + 1 \wedge y < 5$  in  $\phi$ , then we have that  $a_7, a_6, \dots, a_2$  must be 0, because otherwise  $x$  must be greater than or equal to 4. Therefore, we reduce the search space of SAT solving by fixing the values of six Boolean variables, i.e.,  $a_7, a_6, \dots, a_2$ .

*Remark 1.* Although the SAT solver can leverage Boolean Constraint Propagation (BCP) and backtracking to establish and exploit the correlations of Boolean variables automatically, it is unaware of the restriction about the variables before the search. If having an oracle that narrows down the search space of some bit-vector variables, we can reduce the number of Boolean variables to decide, thereby improving the SAT solving performance by reducing unneeded constraint propagation and backtracking.

### 3.2 Variable Ordering

Consider a formula with a set  $N$  of bit-vector variables, each having a value range of size  $k$ . In the worst case, the SAT solver needs to explore  $k^{|N|}$  possible assignments. Like many other search problems, the order in which we select Boolean variables for assignments has an enormous effect on the performance of SAT solving. Specifically, we observe that the data dependence relations in a constraint can be leveraged. The insight here is that the presence of data dependence means that the values of a subset of variables deterministically derive the values of other variables.

*Example 3.2.* Let us consider the constraint  $\phi$  in Example 3.1. According to the discussion in Section 2.2, before SAT solving, the constraint  $\phi$  is translated to a Boolean formula consisting of Boolean variables,  $BV(x)$ ,  $BV(y)$ ,  $BV(x + 1)$ , and  $BV(x * x)$ . In the SAT solving phase, the SAT solver will select and assign values to these Boolean variables in some order. In the example, once  $BV(x)$  are assigned, the values of  $BV(x + 1)$ ,  $BV(y)$ , and  $BV(x * x)$ , can be derived automatically due to the data dependence relations. Thus, giving higher priority to the Boolean variables in  $BV(x)$  than those in  $BV(y)$ ,  $BV(x + 1)$ , and  $BV(x * x)$  can accelerate the speed of cutting the search space.

*Remark 2.* Modern SAT solvers have sophisticated scoring schemes to decide the decision order. However, we find that state-of-the-art schemes like VSIDS are not sufficiently accurate at the beginning of the search. This is because they typically initialize the scores via statistical and syntactical information, which may introduce bias. Therefore, it would be beneficial to assist the solver in the initial

phase by scheduling the decision order according to the semantic information such as the data dependence relations.

## 4 APPROACH

The branching heuristic in SAT solving concerns the order in which the Boolean variables are chosen for assignment and the values assigned to them. As shown in Figure 1(b), our techniques aim to guide it by leveraging two abstract domains over bit-vector variables: the non-relational interval domain and the relational data-dependence domain.

First, we utilize the interval information to fix the values of Boolean variables introduced during bit-blasting, thereby reducing the search space of SAT solving. After that, for variables that cannot be fixed yet, the solver still needs to decide their assignments. Second, to further accelerate SAT solving, we exploit the data-dependence information for guiding the decision order of Boolean variables.

The additional overhead of inferring the information is low: the runtime of the word-level preprocessing phase increases by about 15%. Such a price is acceptable because we can pay a small up-front cost to avoid a large amount of work in the later SAT solving phase. In what follows, we detail the two strategies for improving the performance of SAT solving.

### 4.1 Interval-Guided Variable Assignments

We first present the strategy of using the interval analysis to reduce the search space. Our observation is that in real-world programs, not all statements are complex. Thus, a constraint generated by a program analyzer usually contains many simple sub-constraints, which can be used to infer a sound interval of some bit-vector variables or terms,  $t$ , thereby restricting the values of their corresponding Boolean variables  $BV(t)$ .

However, given a bit-vector formula, acquiring the precise interval information of its variables is non-trivial. Specifically, computing the most precise interval of a variable is NP-hard, which can be reduced to Max-SMT problems [53]. For example, to obtain the maximum value of a bit-vector variable  $x$  subject to a formula  $\phi(x, \dots)$ , we can solve the Max-SMT problem:

$$\begin{cases} \text{maximize} & x \\ \text{subject to} & \phi(x, \dots). \end{cases} \quad (2)$$

This is impractical because solving the optimization problem can be even harder than checking the satisfiability of the formula.

*4.1.1 Interval Analysis.* To balance the precision and performance, we employ a lightweight interval analysis, which takes a bit-vector formula as input, and determines a sound approximation, i.e., interval, of the numeric values for the bit-vector variables in a formula [37]. The details of the interval analysis are omitted, as it is direct and not our key contribution. For instance, given the constraint  $z = x + y$ , where  $x \in [l_x, u_x]$ ,  $y \in [l_y, u_y]$ . According to the rules in the previous work [37], we have  $z \in [l_x + l_y, u_x + u_y]$ . In addition, given a constraint  $z = x + y \wedge z = u - w$ , we need to compute the intervals for  $z = x + y$  and  $z = u - w$ , respectively, and then conjunct the two intervals to get an interval for the variable  $z$ .

Conventionally, let  $R_1 = [l_1, u_1]$  and  $R_2 = [l_2, u_2]$  be two intervals. The logical conjunction and disjunction operations are modeled with the meet (denoted  $\sqcap$ ) and join (denoted  $\sqcup$ ) operations, respectively:

$$\begin{aligned} R_1 \sqcap R_2 &= \begin{cases} \perp & \text{, if } \max(l_1, l_2) > \min(u_1, u_2) \\ [\max(l_1, l_2), \min(u_1, u_2)], & \text{otherwise} \end{cases} \\ R_1 \sqcup R_2 &= [\min(l_1, l_2), \max(u_1, u_2)]. \end{aligned} \quad (3)$$

For example, applying the join operator to the intervals,  $R_1 = [1, 3]$  and  $R_2 = [8, 11]$ , produces a new interval  $R' = [1, 11]$ . Apparently, such a join operation will lose precision as it introduces a set of false values  $\{4, 5, 6, 7\}$  within  $R'$ . In the remainder of this subsection, we focus on how to mitigate such precision loss so that the interval analysis can non-trivially improve the performance of SAT solving.

*Disjunctive Domain with Lazy Join.* One solution to the problem of precision loss is enriching the analysis with disjunction [68, 73]. For instance, instead of computing the new interval  $R'$ , we record the interval as  $R_1 \cup R_2$  in the above example to avoid the precision loss. Unfortunately, the number of intervals to maintain may grow extremely large, which is exponential in the number of disjunctions in the formula.

To restore the precision loss caused by joins but to avoid the expensive disjunctive abstraction, we employ a selective merging heuristic. The heuristic is responsible for determining whether two intervals should be merged with join or kept separately. The basic idea is that, if a join does not introduce much precision loss (i.e., yields a similar set of abstract states as taking their union), we can perform a join. To measure the precision loss quantitatively, we define a *dissimilarity ratio*  $\sigma$  ranging from 0 to 1. If the ratio is smaller than a threshold, we perform a join. Next, we discuss the method for computing the ratio.

First, we need to quantify the dissimilarity between the two intervals. Conventionally, the Hausdorff distance is a common measure of the discrepancy between two polyhedra  $A$  and  $B$ , defined as

$$\begin{aligned} H(X, Y) &= \max_{x \in X} \{\min_{y \in Y} \{d(x, y)\}\} \\ \text{Hausdorff}(A, B) &= \max\{H(A, B), H(B, A)\}, \end{aligned} \quad (4)$$

where  $d(x, y)$  is the Euclidean distance between two points  $x$  and  $y$  in Euclidean space.

The Hausdorff distance between two general polyhedra is hard to compute [6]. Fortunately, in our setting,  $A$  and  $B$  are both intervals. Suppose that we have  $A = [l_1, u_1]$  and  $B = [l_2, u_2]$ . It can be proved that:

$$\text{Hausdorff}(A, B) = \max(|l_1 - l_2|, |u_1 - u_2|). \quad (5)$$

Then, we can measure the proposition of dissimilarities in the new interval introduced after join. Specifically, we define the dissimilarity ratio as

$$\begin{aligned} \sigma &= \text{Hausdorff}(A, B) \div |A \sqcup B| \\ &= \max(|l_1 - l_2|, |u_1 - u_2|) \div (\max(u_1, u_2) - \min(l_1, l_2)). \end{aligned} \quad (6)$$

*Example 4.1.* Consider the example in Figure 2. Suppose that we focus on merging intervals of the variable  $x$ . Comparing Figure 2(a) and Figure 2(b) where  $R_{1x}$  and  $R_{2x}$  are disjoint, we tend to perform a join for the intervals in Figure 2(a) because fewer false positives would be introduced. Comparing Figure 2(c) and Figure 2(d) where

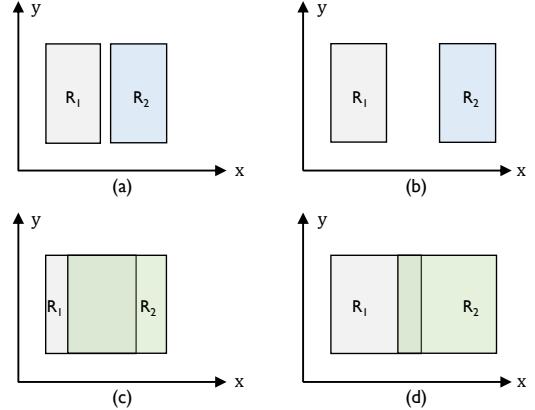


Figure 2: An example of selective merge with join operator.

$R_{1x}$  and  $R_{2x}$  overlap, we tend to perform a join for the intervals in Figure 2(c) because the two intervals have a larger overlap.

*Example 4.2.* Consider the bit-vector formula  $\phi \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_3 \vee \varphi_4)$ , where  $\varphi_1, \varphi_2, \varphi_3$ , and  $\varphi_4$  are all conjunctions of atomic predicates. Suppose that we can infer from  $\varphi_1, \varphi_2, \varphi_3$ , and  $\varphi_4$  that  $R_1 = [3, 10], R_2 = [3, 10], R_3 = [3, 4]$ , and  $R_4 = [8, 11]$ , respectively. We will merge the two intervals  $R_1$  and  $R_2$  with join because the dissimilarity ratio is 0. The merged interval  $R'$  is  $R_1 \sqcup R_2 = [3, 10]$ . In contrast, we tend not to merge  $R_3$  and  $R_4$  because by Eq. (6), the dissimilarity ratio is  $7 \div 8$ , close to 1. Intuitively, the merged interval  $[3, 11]$  would introduce three false positives, i.e.,  $\{5, 6, 7\}$ . Later, we need to propagate the interval  $R'$  to  $R_3$  and  $R_4$ , i.e., compute  $R' \sqcap R_3$  and  $R' \sqcap R_4$ , respectively.

**4.1.2 Reducing Search Space.** After obtaining the sound intervals of bit-vector variables, we then guide the branching heuristic by fixing the values for a subset of Boolean variables. For example, let  $x$  be a bit-vector variable representing a  $n$ -bits unsigned integer such that  $x = [b_{n-1}, \dots, b_0]$ . Recalling Eq. (1), once the values of  $b_{n-1}, b_{n-2}, \dots, b_0$  are fixed, we can obtain the value of  $x$ , and vice versa. Therefore, if the interval analysis infers that  $x \in [l, u]$ , where  $l > 0$  or  $u < 2^n - 1$ , then we fix some Boolean variables from  $BV(x)$ , prior to the main CDCL search loop. If the interval  $[l, u]$  is precise enough, we can produce a dramatic size reduction of the space of the truth assignments searched in by the SAT solver.

An alternative use of the intervals might be adding them as additional constraints to the original formula. However, specifying the constraints can increase the formula size dramatically, when the number of variables is large and each variable has many disjunctive intervals. Consequently, the additional constraints may eventually increase the burden of SAT solving.

## 4.2 Dependence-Guided Variable Ordering

The interval analysis does not provide a panacea: there exist Boolean variables whose values cannot be fixed if the interval information is not precise enough. When solving the Boolean formula generated by bit-blasting, how to effectively handle those variables remains a problem. In particular, as demonstrated in Algorithm 1, the order

to assign Boolean variables decides the search direction and, thus, is crucial for the performance of SAT solving.

To address the challenge, we now describe the strategy that leverages the word-level data-dependence information to guide the decision order of Boolean variables after bit-blasting (§ 4.2.1). We also discuss the combination of our strategy and the VSIDS branching heuristic (§ 4.2.2).

**4.2.1 Ordering with Data Dependence.** As discussed in § 2.2, in addition to encoding a bit-vector variable with a sequence of Boolean variables, the bit-blasting procedure also introduces auxiliary Boolean variables to represent each bit-vector term, i.e., the outcome of calculations over the bit-vector variables. Thus, we need to schedule the decision order for Boolean variables encoding both the bit-vector variables and the terms.

Our key observation is that for a formula with a set  $N$  of variables and terms, the values of a subset  $S$  are often sufficient to determine the values of all variables. Intuitively, this is because program variables and statements usually have some data-dependence relations. For instance, it is common in real-world programs that fresh variables are created for naming expressions, i.e., the calculation results of other variables. Such assignments naturally introduce data dependence.

In the SAT solving phase (Algorithm 1), once the Boolean variables in  $BV(S)$  are assigned, the BCP can infer the values of other variables in  $BV(N \setminus S)$  automatically. Intuitively, at the beginning of the search, the assignments to  $BV(S)$  can cause longer chains of constraint propagation, because the values of other variables, as well as the values of terms over the variables, are the deterministic consequence of that choice.

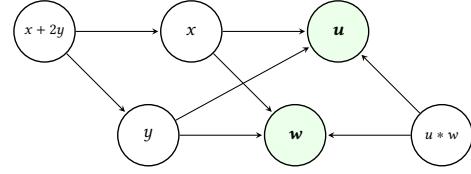
Therefore, our basic idea is to give higher initial scores to the variables in  $BV(S)$ , which has two benefits. First, if the decisions made within  $BV(S)$  do not lead to conflicts, the BCP can infer more truth values for other variables, accelerating the finding of a satisfying model. Second, if some “inappropriate” decisions lead to conflicts, the SAT solver will use the conflicts information to refine the assignments. The refinement can further influence the values of other clauses that are dependent on  $BV(S)$ .

We now discuss how to identify the subset  $S$  and how to prioritize the elements within  $S$ . We start by constructing a data-dependence graph for all bit-vector variables and terms. The graph is split into a set of independent sub-graphs with a formula slicing algorithm [83], such that nodes in each sub-graph must have some data dependence. For example, as shown in Figure 3, a node in the graph represents a variable or a term, and an edge represents a data-dependence relation. Then, we schedule the decision order as follows.

**RULE 1.** Given two nodes  $v_1$  and  $v_2$  such that  $v_2$  is data-dependent on  $v_1$ , we have  $BV(v_2) \leq BV(v_1)$ , meaning that the Boolean variables in  $BV(v_1)$  has a higher priority than those in  $BV(v_2)$ .

Some clients of SMT solvers, such as symbolic execution, may eliminate intermediate program variables that name bit-vector terms, such that the formulas mention only input variables. In such cases, our approach can still leverage the data dependence to guide the decision order.

**Example 4.3.** Consider the constraint in Figure 3. Suppose that the variables  $x$  and  $y$  are the intermediate variables while the others



**Figure 3: Data-dependence graph for the constraint**  $\phi \equiv x = u + w \wedge y = 2 * u - w \wedge x + 2 * y < 10 \wedge u * w < 60$ .

are input variables of a program. A symbolic executor may generate a constraint where the variables  $x$  and  $y$  are replaced by  $u + w$  and  $2 * u - w$ , respectively, so that the constraint only contains program input variables:

$$\phi \equiv u + w + 2 * (2 * u - w) < 10 \wedge u * w < 60.$$

We denote the terms,  $u+w$ ,  $2*u-w$ ,  $u*w$ , and  $u+w+2*(2*u-w)$ , as,  $t_1$ ,  $t_2$ ,  $t_3$ , and  $t_4$ , respectively. As illustrated in Example 2.2, the bit-blasted needs to introduce Boolean variables to encode each element in  $u$ ,  $w$ ,  $t_1$ ,  $t_2$ ,  $t_3$ , and  $t_4$ . Since all the terms are data-dependent on  $u$  and  $w$ , we have

$$BV(t_1), BV(t_2), BV(t_3), BV(t_4) \leq BV(u), BV(w).$$

Meanwhile, we have  $BV(t_4) \leq BV(t_1)$  and  $BV(t_4) \leq BV(t_2)$  because  $t_4$  is data-dependent on  $t_1$  and  $t_2$ .

In addition to data dependence, control dependence relations encoded in a constraint also can be leveraged to order the nodes that do not have data dependence relations. For instance, for a simple program `if (c) { v = a; } else { v = b; }`, where the value of the variable  $v$  is control-dependent on the condition  $c$ , if the value of  $c$  is known, we then only need to compute the value of either  $a$  or  $b$ . Thus, when solving a constraint encoding this control-dependence relation, it is preferable to have  $BV(a), BV(b) \leq BV(c)$ . In practice, such a control-dependence relation is usually encoded as an *ite* (if-then-else) constraint:  $\phi \equiv v = \text{ite}(c, a, b)$ , or its equivalent forms. Given such a constraint, we have the following rule.

**RULE 2.** Given a formula in the form of  $v = \text{ite}(c, a, b)$  or its other equivalent forms, we have  $BV^*(a), BV^*(b) \leq BV^*(c)$ , where we use  $BV^*(t)$  to represent the Boolean variables in  $BV(t)$  and all other variables the term  $t$  data-depends on.

**4.2.2 In Combination with VSIDS.** In principle, our strategy can replace the default VSIDS heuristic, which maintains a score for each variable and updates the scores periodically (§ 2.3). However, in practice, relying exclusively on the dependence-based scheme is not practical because it cannot dynamically refine the scores. The previous study [12] shows that branching strategies with dynamic re-ordering are usually more effective than static ones.

Therefore, we combine our scoring scheme with VSIDS for the decision-making. Recall that VSIDS initializes the variable scores based on only statistical information, such as the number of clauses in which a variable appears. The previous work [56, 85] has shown that guiding VSIDS in initializing the scores can notably increase the efficiency of the solving process. Hence, we initialize the scores by combining the dependence and the statistical information, and then update the scores by following VSIDS.

In general, the impact of the data-dependence information decreases over time because VSIDS favors the most recently detected conflict clauses, which will eventually dominate when the search progresses. Fortunately, in practice, VSIDS can make more informed decisions when the search progresses, because it can gather more information about the search history to make a sophisticated choice.

To summarize, our design aims to guide the SAT solver at the beginning of the search, when VSIDS is not yet well-formed enough, and allows VSIDS to override the initial decisions when it has a deeper understanding of the formula.

## 5 EVALUATION

We implement TRIDENT on top of the Z3 SMT solver. Specifically, TRIDENT uses Z3 for parsing formulas in the SMT-LIB v2.6 format, performing word-level simplifications, and conducting the bit-blasting. Finally, the translated Boolean formulas are handed to our customized SAT solver, which leverage our interval and data-dependence analyses to guide its branching heuristics. Our evaluation is designed to answer the following research questions:

- **RQ1:** How effective are the two guidance strategies of TRIDENT (§ 5.2)?
- **RQ2:** Is TRIDENT faster to solve constraints compared to other state-of-the-art SMT solvers (§ 5.3)?
- **RQ3:** Can TRIDENT improve the scalability of existing symbolic analysis tools (§ 5.4)?

### 5.1 Experimental Setup

*Subjects.* We use three realistic symbolic analysis tools to generate bit-vector constraints for evaluating our approach. Specifically, ANGR [78]<sup>4</sup> is a binary analysis platform, QSYM [88]<sup>5</sup> is a symbolic execution engine for hybrid fuzzing, and PINPOINT [77] is a path-sensitive static bug finder. We target subjects that cover different scales of programs, whose sizes range from a few thousand lines to multi-million lines, cover a wide range of applications, such as networking libraries and database engines, and cover both standard benchmarks and open-source projects. In total, as listed in Table 2, we collect thirty programs in three groups:

- ANGR is configured to analyze ten programs from Coreutils,<sup>6</sup> a commonly-used dataset in symbolic execution.
- The ten programs run by QSYM are taken from the tool's original paper [88], including three programs in the LAVA-M dataset [30] and seven open-source projects.
- We apply PINPOINT to analyze ten industrial-sized C/C++ programs, which are the monthly trending projects on GitHub that we can set up.

To answer RQ1 and RQ2, we run the tools with their default solvers to dump SMT queries as the SMT-LIB v2.6 format, and then conduct the experiments on these queries. The number of total queries generated by ANGR, QSYM, and PINPOINT is 2,123,211, 1,502,958, and 3,806,989, respectively. Among the queries, 5,236,735 instances are satisfiable, and 2,196,423 ones are unsatisfiable. We believe such a large number of instances are sufficient to evaluate the performance of SMT solving.

<sup>4</sup><https://github.com/angr/angr>

<sup>5</sup><https://github.com/sslab-gatech/qsym>

<sup>6</sup><https://www.gnu.org/software/coreutils>

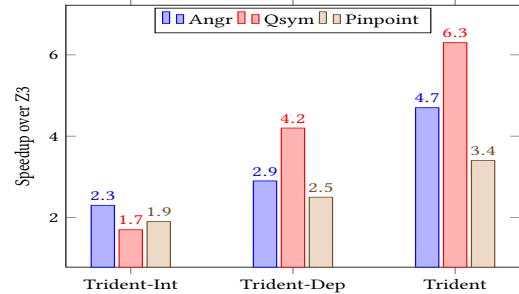


Figure 4: Relative performance impact of the strategies.

Table 1: Number of unsolved queries.

Group	Z3	TRIDENT-INT	TRIDENT-DEP	TRIDENT
ANGR	61	29	27	23
QSYM	133	121	32	11
PINPOINT	57	45	21	21

To answer RQ3, we substitute TRIDENT for Z3, which is the SMT solver used by ANGR, QSYM, and PINPOINT. Since TRIDENT retains the Z3 API, the program analyzers using Z3 can directly benefit from our work. In the experiment, we examine whether the superior performance of TRIDENT translates into end-to-end benefits for the symbolic analysis tools.

*Environment.* All solvers are compiled with gcc 7.4.0 using the flags `-O3 -m64 -march=native`. We configure each solver to use a per-query timeout of 30 seconds, following the prior works [66, 71] on accelerating SMT solving in symbolic analysis. All solvers are compiled with gcc 7.4.0 using the flags `-O3 -m64 -march=native`. We configure each solver to use a per-query timeout of 15 seconds.

### 5.2 RQ1: Effectiveness of the Guidance Strategies

First, we investigate the effectiveness of the strategies in our solver by comparing its four configurations. Specifically, we compare TRIDENT to the original Z3 solver,<sup>7</sup> TRIDENT-INT, and TRIDENT-DEP, two configurations with each of the interval-guided assignment or dependence-guided ordering strategies turned on.

Figure 4 shows the results in terms of solving time. The data for each benchmark group is normalized to the runtime of Z3. A number larger than 1.0 is a speedup. Table 1 compares the number of unsolved queries within the given time limit.

We can observe that both of the two strategies in TRIDENT contribute to its performance. The data dependence-based strategy has better effects than that of interval-based strategy. It would be hard to infer precise interval information for some queries, rendering the results less effective in reducing the search space. In the QSYM group, data-dependence analysis is the most effective. We find that queries from QSYM tend to have many variables that have strong

<sup>7</sup>We set the SAT solver of Z3 to use the VSIDS branching heuristic and the Luby restart strategy [55].

data-dependence relations. Overall, the speedups range from  $3.4\times$  to  $6.3\times$ .

Additionally, using the two optimizing strategies, TRIDENT is able to solve 38, 122, and 36 more queries than Z3 in the ANGR, QSYM, and PINPOINT groups, respectively.

TRIDENT leverages the interval and data-dependence information to reduce the solving time of Z3 by  $3.4\times$  to  $6.3\times$ , as well as increase the number of solved queries.

### 5.3 RQ2: Comparison to Other SMT Solvers

In addition to comparing with Z3, on which TRIDENT is built, we also examine the practicality of TRIDENT by comparing it against two state-of-the-art SMT solvers, namely, CVC4 v1.7 and BOOLECTOR v2.4.1. In particular, BOOLECTOR won the first place in the QF\_BV track of SMT-COMP 2019,<sup>8</sup> and CVC4 also won many champions in other tracks.<sup>9</sup>

Figure 5 plots the cumulative runtime of the solvers. Table 2 shows the detailed comparison results. For each program, we report the total SMT queries, the number of unsolved queries, and the time cost of the solvers. The last column of Table 2 denotes the speedup achieved by TRIDENT over the fastest baseline in CVC4 and BOOLECTOR.

We can notice that, for most of the programs, TRIDENT obtains  $2.0\times$  to  $3.1\times$  increases in performance against the baseline. For the ANGR, QSYM, and PINPOINT groups, TRIDENT is, on average  $3.0\times$ ,  $2.3\times$ , and  $3.1\times$  faster than the baseline, respectively. The largest improvements are seen for ffmpeg and v8 in the PINPOINT group, with speedups of  $4.2\times$ .

There are four programs for which the speedups are smaller than  $2.0\times$ , including nm, tcpdump and jhead in the QSYM group, and mysql in the PINPOINT group. However, we have seen other solvers much slower when handling queries from several programs. When solving queries from who, size, tcpdump, and jhead, CVC4 is more than  $11\times$  slower than TRIDENT. In the cases of gluster, ffmpeg, and v8, BOOLECTOR is  $4\times$  slower than TRIDENT. To sum up, TRIDENT is, on average  $7.9\times$  and  $2.9\times$  faster than CVC4 and BOOLECTOR, respectively.

There are a total of eleven programs for which the three SMT solvers finish all queries: six of them are in the ANGR group, and five of them are in the QSYM group. In total, TRIDENT solves 6, 2, and 40 more queries than the baseline for the ANGR, QSYM, and PINPOINT groups, respectively. Overall, we conclude that TRIDENT can solve as many queries as the other two solvers.

Compared with the fastest solver between CVC4 and BOOLECTOR, TRIDENT improves the SMT solving speed by  $1.6\times$  to  $4.2\times$ , without lessening the number of solved queries within the time limit.

<sup>8</sup><https://smt-comp.github.io/>

<sup>9</sup><https://cvc4.github.io/awards.html>

### 5.4 RQ3: Improving the Scalability of Symbolic Analysis Tools

Finally, we evaluate the usefulness of TRIDENT for three program analysis clients. Specifically, we configure ANGR for running control-flow recovery analysis [78], QSYM for generating test cases, and PINPOINT for detecting null pointer dereference bugs. For each client, we measure the speedups achieved by the three configurations of our solver, following the settings of § 5.2.

*Results of ANGR.* Figure 6 shows the reduction of analysis time for control-flow recovery, which is a fundamental step for binary analysis [24, 74, 78]. On average, TRIDENT-INT, TRIDENT-DEP, and TRIDENT demonstrate  $1.5\times$ ,  $1.6\times$ , and  $1.9\times$  speedups, respectively. The overall analysis speedups are smaller than the pure SMT solving speedups, because, in addition to the symbolic execution engine, ANGR consists of other components that can be time-consuming, such as the backward slicing [78].

*Results of QSYM.* Figure 7 summarizes the speedup of input generation in 24 hours of testing. To sum up, on average, TRIDENT-INT, TRIDENT-DEP, and TRIDENT increase the speed by  $1.2\times$ ,  $1.5\times$ , and  $1.6\times$  speedups, respectively. As a case study, Figure 8 shows the cumulative branch coverage for objdump and readelf. The speedups are smaller than the pure SMT solving speedups because QSYM is used for hybrid fuzzing [82, 88], where a significant proportion of the time is spent on program execution and symbolic emulation. Overall, the results show that TRIDENT significantly improves the scalability of QSYM, a state-of-the-art symbolic execution engine for test case generation.

*Results of PINPOINT.* In Table 3, we report speedups on the four largest programs, which are representative of speedups on the remaining programs. On average, TRIDENT-INT, TRIDENT-DEP, and TRIDENT demonstrate about  $1.5\times$ ,  $1.7\times$ , and  $2.4\times$  speedups, respectively. We remark that the time reduction is non-trivial for a static bug finder, which is enough to make an originally impractical analysis usable in practice. For example, originally, PINPOINT cannot finish analyzing v8 within 11.4 hours. In contrast, with the optimizations offered by TRIDENT, PINPOINT completes the analysis within 5 hours, meaning that it can run in nightly mode, i.e., perform the full analysis of a large codebase from scratch, run by the continuous integration system over night [13].

On average, TRIDENT increases the end-to-end analysis speed of three clients in ANGR, QSYM, and PINPOINT by  $1.9\times$ ,  $1.6\times$ , and  $2.4\times$ , respectively.

### 5.5 Threats to Validity

The internal validity mainly depends on the correctness of our implementation. To reduce this threat, at least four developers review the source code of TRIDENT. Furthermore, we have performed extensive testing of TRIDENT on thousands of applications, during which it has solved billions of SMT queries. For these queries, we compare the solving results given by TRIDENT and several existing SMT solvers for validating the correctness of our implementation.

The threats to external validity lie in our test suits. To reduce the threat resulting from benchmarks, we validate our approach

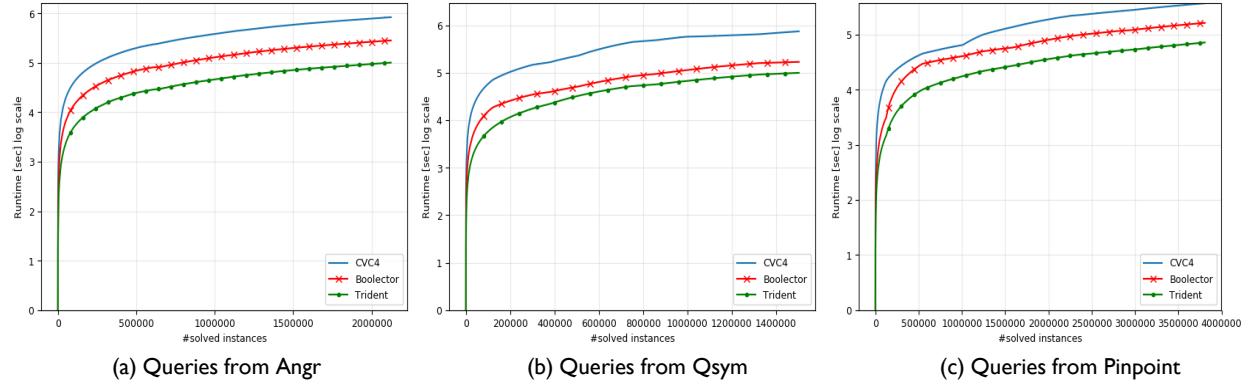


Figure 5: Results of CVC4, BOOLECTOR, and TRIDENT on all solved instances.

Table 2: Results of the SMT solvers over the test suites. For each solver, we record the number of unsolved queries and the total time (in minutes) taken. For TRIDENT, the column “Speedup” denotes the speedup over the *fastest* solver between CVC4 and BOOLECTOR.

Group	Program	Queries	CVC4		BOOLECTOR		TRIDENT		
			TO	Time	TO	Time	TO	Time	Speedup
ANGR	fmt	363,552	8	2598	3	862	3	<b>279</b>	3.1×
	logname	189,243	0	1167	0	366	0	<b>119</b>	3.1×
	mkfifo	146,998	0	1120	0	342	0	<b>110</b>	3.3×
	nice	307,243	62	2214	8	736	9	<b>238</b>	3.1×
	nohup	140,296	0	859	0	267	0	<b>87</b>	3.1×
	env	253,595	21	1708	5	585	0	<b>189</b>	3.1×
	head	97,438	0	642	0	212	0	<b>68</b>	2.7×
	mv	173,122	20	1060	13	335	11	<b>109</b>	3.1×
	nl	145,582	0	779	0	191	0	<b>70</b>	2.7×
	nproc	306,142	0	1850	0	586	0	<b>191</b>	3.1×
QSYM	uniq	133,587	0	1211	0	243	0	<b>121</b>	2.0×
	md5sum	92,692	0	188	0	188	0	<b>59</b>	3.2×
	who	121,647	0	1001	9	259	0	<b>90</b>	2.9×
	readelf	113,812	11	726	0	172	6	<b>77</b>	2.2×
	nm	80,496	0	327	0	71	0	<b>40</b>	1.8×
	objdump	178,555	0	1053	0	209	0	<b>88</b>	2.4×
	size	129,580	19	919	1	160	5	<b>71</b>	2.3×
	tcpdump	236,256	0	3536	2	521	0	<b>292</b>	1.8×
	djpeg	235,357	0	970	1	525	0	<b>226</b>	2.3×
	jhead	180,976	0	1383	0	118	0	<b>76</b>	1.6×
PINPOINT	gluster	465,054	1	211	28	411	0	<b>65</b>	3.2×
	libicu	226,418	2	226	0	134	0	<b>50</b>	2.7×
	imagemagick	111,122	4	89	2	54	1	<b>22</b>	2.5×
	openssl	356,173	28	219	6	155	1	<b>44</b>	3.5×
	python	124,685	3	72	0	53	0	<b>19</b>	2.8×
	gcc	132,550	13	108	0	47	0	<b>23</b>	2.0×
	ffmpeg	419,306	91	416	2	402	1	<b>95</b>	4.2×
	v8	645,970	100	674	7	616	4	<b>146</b>	4.2×
	mysql	468,879	28	198	6	133	6	<b>70</b>	1.9×
	wine	856,832	150	677	10	719	8	<b>187</b>	3.6×
<b>Total TO</b>			561		103		55		

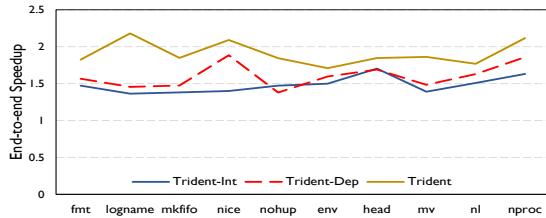


Figure 6: Speedup of control-flow recovery analysis in ANGR.

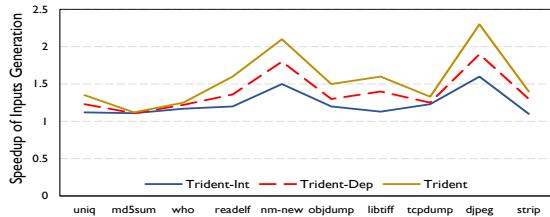


Figure 7: Speedup of test case generation using QSYM.

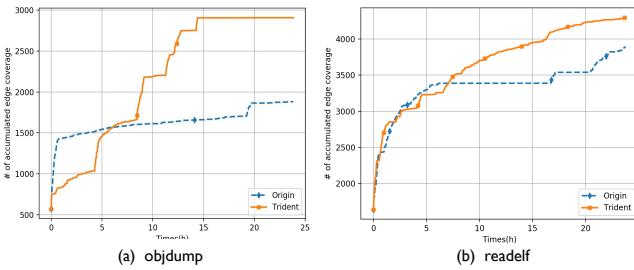


Figure 8: Number of covered branches in 24 hours.

Table 3: Reducing the end-to-end bug finding time (in minutes) of PINPOINT.

Program	KLoC	Z3	TRIDENT-INT	TRIDENT-DEP	TRIDENT
ffmpeg	1001	315	207 (1.5×)	188 (1.7×)	118 (2.7×)
v8	1201	687	424 (1.6×)	359 (1.9×)	278 (2.5×)
mysql	2030	164	122 (1.3×)	112 (1.5×)	97 (1.7×)
wine	4108	526	302 (1.7×)	329 (1.6×)	210 (2.5×)

over three different symbolic analysis platforms and on thirty real-world subjects varying in scale and functionality. However, the queries from these platforms are not necessarily representative of other tools. Another threat to validity is whether our strategies can be generalized to other eager SMT solvers. In the future, we will further apply our approach to other solvers and more symbolic analysis frameworks.

## 6 RELATED WORK

We discuss related work in three groups: symbolic execution (§ 6.1), bit-vector constraint solving (§ 6.2) and branching heuristics (§ 6.3).

## 6.1 Accelerating SMT Solving in Symbolic Execution

There is a large body of work on accelerating constraint solving for symbolic execution. Different forms of caches such as satisfying model [20] and unsatisfiable core [60] help avoid calling the solver when possible. KLEE employs a method called “constraint independence” to eliminate sub-formulas that are irrelevant to the current branch expression. Our approach can guide the SAT solver to better handle variables that are not independent, by leveraging their data-dependence relations. Symbolic executors can perform different algebraic simplifications of expressions to make the constraints more solver-friendly [21, 67, 76]. Semantics-preserving program transformations [19, 31, 84] also help to generate simpler and fewer SMT queries. These simplifications and transformations operate at the level of the symbolic execution engines, and, thus, are orthogonal to our work that operates at the level of SMT solvers. The idea of employing a portfolio of SMT solvers for parallel solving has been explored [66], whereby each solver runs independently, and the result of the fastest solver is taken. Our solver can be combined with existing solvers in a portfolio.

## 6.2 Decision Procedures for Bit-Vectors

The majority of the state-of-the-art SMT solvers [2, 25, 29, 33, 63] solve bit-vector constraints via reduction to SAT, some employing specialized procedures for equality reasoning [17] and linear modulo arithmetic [36]. A number of methods alternative to bit-blasting have been developed, such as the Canonizer-based approaches [4, 28], reduction to Effectively Propositional Logic (EPR) [49], the model-constraining satisfiability calculus (mcSAT) [89], the stochastic local search (SLS) algorithms [35, 64, 65], and the abstraction-based methods [18]. Most of these works operate over restricted subsets of bit-vector theory. We tried the SLS-based solvers in BOOLECTOR and Z3, but found that they are not competitive with the solvers examined in our evaluation.

Most existing works on enhancing the eager approach focus on improving the word-level preprocessing and bit-blasting phases [34, 45, 48, 57, 70, 81]. Unconstrained term propagation [16, 17] identifies variables and terms that are irrelevant for determining satisfiability, thereby reducing the problem size. Nadel [61] proposes an algorithm that generates word-level rewriting rules at runtime for a given problem. Singh and Solar-Lezama [81] generates word-level simplifiers using program synthesis. Inala et al. [45] generate domain-specific encoding schema for bit-blasting with synthesis and machine learning techniques. These optimizations for word-level preprocessing and bit-blasting are orthogonal to our approach.

Interval Constraint Propagation (ICP) is a sound but incomplete numerical method for constraint solving [7, 8, 14, 23, 32, 38]. Most existing ICP techniques work only for real or integer formulas. Janota and Wintersteiger [46] propose a method for inferring interval information of bit-vectors from a system of simple inequalities. In each of the inequality, only one variable is permitted and there are no multiplications. Dustmann et al. [32] present a semi-decision procedure for bit-vectors, which tracks the possible values for each symbolic variable, so as to quickly solve certain types of queries before bit-blasting. Compared with previous works, our analysis is not restricted to certain classes of queries. Besides, it attempts to

reduce the search space of SAT solving after bit-blasting, instead of directly deciding the satisfiability.

### 6.3 Branching Heuristics

There is a huge literature on SAT branching heuristics, such as MOM [69], Jeroslow-Wang [47], VSIDS [59], Conflict History-Based (CHB) [54], and so on. Different variants of VSIDS have also been proposed, such as BerkMin’s strategy [41], exponential VSIDS (EVSIDS) [10], variable move-to-front (VMTF) [72], clause-move-to-front (CMTF) [39], and average conflict index decision score (ACIDS) [12]. However, these branching heuristics are designed for general SAT problems.

The dependence information has been used to restrict the set of decision variables for SAT solving [26, 27, 52, 58]. Given a Boolean formula with a set  $N$  of variables, they restrict the branching heuristic to focus on a subset  $S$  that is sufficient to determine the truth values of all variables. In contrast, we leverage the data-dependence relations to initialize the scores, without restricting the branching only to the subset. Besides, we combine the data- and control-dependence information to further schedule the order of variables within the subset  $S$ .

Previous works show that structure information can be utilized to improve the branching heuristic for solving SAT queries from bounded model checking (BMC). Shtrichman [79] predetermines the variable ordering by traversing the variable dependency graph. Wang et al. [85] identify important variables from previous unsatisfiable BMC instances and apply them to solve the current instance. Yin et al. [87] give a higher priority to the transition variables from the transition system. These approaches target pure Boolean formulas and leverage the domain-specific knowledge of circuits, instead of word-level structure and information.

The theory-aware approaches [9, 22, 42, 75] for DPLL( $T$ ) lazy SMT solving attempt to make the SAT branching heuristic aware of the  $T$ -semantic of the literals. Z3STR3 [9] biases the search towards branches that contain easier string constraints. Goldwasser et al. [42] guide the branching by leveraging the  $T$ -implications between linear arithmetic constraints. Being aware of the implication relations, they can choose unassigned atomic predicates that are consistent with the current partial assignment. A recent work [22] takes advantage of the control-flow information to reduce the redundant search space. These methods differ from our method across two key technical dimensions. First, they work under the context of lazy SMT solving, while our approach attempts to accelerate eager SMT solving. Given a first-order formula, our reasoning centers around the bit-blasted and equisatisfiable Boolean formula, but not around the approximated Boolean skeleton. Second, they exploit correlations between atomic predicates locally, while we analyze globally word-level information such as the variable interval.

## 7 CONCLUSION

This paper presents an approach to accelerating eager bit-vector solvers, which infers the interval and data-dependence information of the bit-vector formula to guide the SAT solver after bit-blasting. We have evaluated the proposed techniques over queries from three realistic symbolic analysis platforms, demonstrating the advantages of our approach.

## ACKNOWLEDGMENTS

We thank Yushan Zhang, Yikun Hu, as well as the anonymous reviewers for their insightful comments. This work is partially funded by an MSRA grant, the Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, ITS/440/18FP grants, and the NSFC Project No. 61902329. Qingkai Shi is the corresponding author.

## REFERENCES

- [1] Athanasios Avgerinos. 2014. *Exploiting Trade-offs in Symbolic Execution for Identifying Security Bugs*. Ph.D. Dissertation.
- [2] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) (*CAV’11*). Springer-Verlag, Berlin, Heidelberg, 171–177.
- [3] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The satisfiability modulo theories library (SMT-LIB). [www.SMT-LIB.org](http://www.SMT-LIB.org) 15 (2010), 18–52.
- [4] Clark W Barrett, David L Dill, and Jeremy R Levitt. 1998. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference* (San Francisco, California, USA) (*DAC ’98*). ACM, New York, NY, USA, 522–527. <https://doi.org/10.1145/277044.277146>
- [5] Clark W. Barrett, David L. Dill, and Aaron Stump. 2002. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification* (*CAV ’02*). Springer-Verlag, Berlin, Heidelberg, 236–249. <https://doi.org/10.5555/647771.734410>
- [6] Michael Bartovn, Iddo Hanniel, Gershon Elber, and Myung-Soo Kim. 2010. Precise Hausdorff Distance Computation between Polygonal Meshes. *Comput. Aided Geom. Des.* 27, 8 (Nov. 2010), 580–591. <https://doi.org/10.1016/j.cagd.2010.04.004>
- [7] Jason Belt, Robby, and Xianghua Deng. 2009. Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-Based Analyses. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) (*ESEC/FSE ’09*). Association for Computing Machinery, New York, NY, USA, 355–364. <https://doi.org/10.1145/1595696.1595762>
- [8] Frédéric Benhamou and Laurent Granvilliers. 2006. Continuous and interval constraints. In *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 571–603.
- [9] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3STR3: A String Solver with Theory-Aware Heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design* (Vienna, Austria) (*FMCAD ’17*). FMCAD Inc, Austin, Texas, 55–59. <https://doi.org/10.5555/3168451.3168468>
- [10] Armin Biere. 2008. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing* (Guangzhou, China) (*SAT’08*). Springer-Verlag, Berlin, Heidelberg, 28–33.
- [11] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD. <https://doi.org/10.5555/1550723>
- [12] Armin Biere and Andreas Fröhlich. 2015. Evaluating CDCL Variable Scoring Schemes. In *Theory and Applications of Satisfiability Testing – SAT 2015*, Marijn Heule and Sean Weaver (Eds.). Springer International Publishing, Cham, 405–422. [https://doi.org/10.1007/978-3-319-24318-4\\_29](https://doi.org/10.1007/978-3-319-24318-4_29)
- [13] Sam Blackshear, Nikos Gorogiannis, Peter W. O’Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. <https://doi.org/10.1145/3276514>
- [14] Mateus Borges, Marcelo d’Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. 2012. Symbolic Execution with Interval Solving and Meta-Heuristic Search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST’12)*. IEEE Computer Society, USA, 111–120. <https://doi.org/10.1109/ICST.2012.91>
- [15] Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media.
- [16] Robert Brummayer. 2009. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. Ph.D. Dissertation. Informatik, Johannes Kepler University.
- [17] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Paliwal, and Roberto Sebastiani. 2007. A lazy and layered SMT (BV) solver for hard industrial verification problems. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) (*CAV’07*). Springer-Verlag, Berlin, Heidelberg, 547–560.
- [18] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. 2007. Deciding Bit-Vector Arithmetic with Abstraction. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal) (*TACAS’07*). Springer-Verlag, Berlin, Heidelberg, 358–372.
- [19] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*

- (Bergamo, Italy) (*ESEC/FSE 2015*). Association for Computing Machinery, New York, NY, USA, 906–909. <https://doi.org/10.1145/2786805.2803205>
- [20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) (*OSDI'08*). USENIX Association, Berkeley, CA, USA, 209–224.
- [21] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2006. EXE: automatically generating inputs of death. (2006), 322–335. <https://doi.org/10.1145/1180405.1180445>
- [22] Jianhui Chen and Fei He. 2018. Control Flow-guided SMT Solving for Program Verification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) (*ASE 2018*). ACM, New York, NY, USA, 351–361. <https://doi.org/10.1145/3238147.3238218>
- [23] Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. 2017. Sharpening constraint programming approaches for bit-vector theory. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 3–20.
- [24] Cristina Cifuentes and Mike Van Emmerik. 1999. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the 7th International Workshop on Program Comprehension* (*IWPC '99*). IEEE Computer Society, USA, 192. <https://doi.org/10.5555/520033.858247>
- [25] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) (*TACAS'13*). Springer-Verlag, Berlin, Heidelberg, 93–107. [https://doi.org/10.1007/978-3-642-36742-7\\_7](https://doi.org/10.1007/978-3-642-36742-7_7)
- [26] Fady Copty, Limor Fix, Ranjan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. 2001. Benefits of Bounded Model Checking at an Industrial Setting. In *Proceedings of the 13th International Conference on Computer Aided Verification* (*CAV '01*). Springer-Verlag, London, UK, UK, 436–453.
- [27] James M. Crawford and Andrew B. Baker. 1994. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 2)* (Seattle, Washington, USA) (*AAAI'94*). American Association for Artificial Intelligence, Menlo Park, CA, USA, 1092–1097. <http://dl.acm.org/citation.cfm?id=199480.199540>
- [28] David Cyrluk, Oliver Möller, and Harald Rueß. 1997. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proceedings of the 9th International Conference on Computer Aided Verification* (*CAV '97*). Springer-Verlag, London, UK, UK, 60–71. <http://dl.acm.org/citation.cfm?id=647766.733602>
- [29] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) (*TACAS'08/ETAPS'08*). Springer-Verlag, Berlin, Heidelberg, 337–340.
- [30] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, 110–121. <https://doi.org/10.1109/SP.2016.15>
- [31] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the Influence of Standard Compiler Optimizations on Symbolic Execution. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE) (ISSRE '15)*. IEEE Computer Society, USA, 205–215. <https://doi.org/10.1109/ISSRE.2015.7381814>
- [32] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: A Multi-interval Theory Solver for Symbolic Execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) (*ASE 2018*). ACM, New York, NY, USA, 430–440. <https://doi.org/10.1145/3238147.3238179>
- [33] Bruno Dutertre. 2014. Yices2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 737–744. [https://doi.org/10.1007/978-3-319-08867-9\\_49](https://doi.org/10.1007/978-3-319-08867-9_49)
- [34] Anders Franzen. 2010. Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT. Ph.D. Dissertation. University of Trento.
- [35] Andreas Fröhlich, Armin Biere, Christoph M Wintersteiger, and Youssef Hamadi. 2015. Stochastic Local Search for Satisfiability Modulo Theories.. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) (*AAAI'15*). AAAI Press, 1136–1143. <http://dl.acm.org/citation.cfm?id=2887007.2887165>
- [36] Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) (*CAV'07*). Springer-Verlag, Berlin, Heidelberg, 519–531.
- [37] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2015. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 1 (Jan. 2015), 35 pages. <https://doi.org/10.1145/2651360>
- [38] Sicun Gao, Malay Ganai, Franjo Ivančić, Aarti Gupta, Sriram Sankaranarayanan, and Edmund M. Clarke. 2010. Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Lugano, Switzerland) (*FMCAD '10*). FMCAD Inc, Austin, Texas, 81–90.
- [39] Roman Gershman and Ofer Strichman. 2005. HAIFASAT: A New Robust SAT Solver. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing* (Haifa, Israel) (*HVC'05*). Springer-Verlag, Berlin, Heidelberg, 76–89. [https://doi.org/10.1007/11678779\\_6](https://doi.org/10.1007/11678779_6)
- [40] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (*PLDI '05*). ACM, New York, NY, USA, 213–223. <https://doi.org/10.1145/1065010.1065036>
- [41] E. Goldberg and Y. Novikov. 2002. BerkMin: A Fast and Robust Sat-Solver. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*. IEEE Computer Society, USA, 142. <https://doi.org/10.5555/882452.874512>
- [42] Dan Goldwasser, Ofer Strichman, and Shai Fine. 2008. A Theory-Based Decision Heuristic for DPLL(T). In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Portland, Oregon) (*FMCAD '08*). IEEE Press, Article 13, 8 pages.
- [43] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. 2014. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 680–695. [https://doi.org/10.1007/978-3-319-08867-9\\_45](https://doi.org/10.1007/978-3-319-08867-9_45)
- [44] Trevor Alexander Hansen. 2012. *A constraint solver and its application to machine code test generation*. Ph.D. Dissertation.
- [45] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. 2016. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2016*, Nadia Creignou and Daniel Le Berre (Eds.). Springer International Publishing, Cham, 302–320. [https://doi.org/10.1007/978-3-319-40970-2\\_19](https://doi.org/10.1007/978-3-319-40970-2_19)
- [46] Mikolás Janota and Christoph M Wintersteiger. 2016. On Intervals and Bounds in Bit-vector Arithmetic.. In *SMT@IJCAR*, 81–84.
- [47] Robert G Jeroslow and Jinchang Wang. 1990. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence* 1, 1–4 (1990), 167–187.
- [48] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. 2009. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) (*CAV '09*). Springer-Verlag, Berlin, Heidelberg, 668–674. [https://doi.org/10.1007/978-3-642-02658-4\\_53](https://doi.org/10.1007/978-3-642-02658-4_53)
- [49] Gergely Kovácsnai, Andreas Fröhlich, and Armin Biere. 2013. bv2Epr: A tool for polynomially translating quantifier-free bit-vector formulas into EPR. In *Proceedings of the 24th International Conference on Automated Deduction* (Lake Placid, NY) (*CADE'13*). Springer-Verlag, Berlin, Heidelberg, 443–449. [https://doi.org/10.1007/978-3-642-38574-2\\_32](https://doi.org/10.1007/978-3-642-38574-2_32)
- [50] Gergely Kovácsnai, Andreas Fröhlich, and Armin Biere. 2016. Complexity of Fixed-Size Bit-Vector Logics. *Theor. Comp. Sys.* 59, 2 (Aug. 2016), 323–376. <https://doi.org/10.1007/s00224-015-9653-1>
- [51] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View* (1 ed.). Springer Publishing Company, Incorporated.
- [52] Chu Min Li. 2000. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, 291–296. <https://doi.org/10.5555/647288.760210>
- [53] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) (*POPL '14*). ACM, New York, NY, USA, 607–618. <https://doi.org/10.1145/2535838.2535857>
- [54] Jie Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. 2016. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) (*AAAI'16*). AAAI Press, 3434–3440. <http://dl.acm.org/citation.cfm?id=3016100.3016385>
- [55] Michael Luby, Alastair Sinclair, and David Zuckerman. 1993. Optimal Speedup of Las Vegas Algorithms. *Inf. Process. Lett.* 47, 4 (Sept. 1993), 173–180. [https://doi.org/10.1016/0020-0190\(93\)90029-9](https://doi.org/10.1016/0020-0190(93)90029-9)
- [56] Zoltán Ádám Mann and Pál András Papp. 2017. Guiding SAT solving by formula partitioning. *International Journal on Artificial Intelligence Tools* 26, 04 (2017), 1750011.
- [57] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. 2012. Automated Reencoding of Boolean Formulas. In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing* (Haifa, Israel) (*HVC'12*). Springer-Verlag, Berlin, Heidelberg, 102–117. [https://doi.org/10.1007/978-3-642-39611-3\\_14](https://doi.org/10.1007/978-3-642-39611-3_14)
- [58] Fabio Massacci and Laura Marraro. 2000. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning* 24, 1–2 (2000), 165–203.
- [59] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the*

- 38th Annual Design Automation Conference* (Las Vegas, Nevada, USA) (DAC '01). ACM, New York, NY, USA, 530–535. <https://doi.org/10.1145/378239.379017>
- [60] Jan Mrázek, Martin Jonás, Vladimír Štill, Henrich Lauko, and Jiří Barnat. 2017. Optimizing and Caching SMT Queries in SymDIVINE. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*. Springer-Verlag, Berlin, Heidelberg, 390–393. [https://doi.org/10.1007/978-3-662-54580-5\\_29](https://doi.org/10.1007/978-3-662-54580-5_29)
- [61] Alexander Nadel. 2014. Bit-Vector Rewriting with Automatic Rule Generation. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 663–679. [https://doi.org/10.1007/978-3-319-08867-9\\_44](https://doi.org/10.1007/978-3-319-08867-9_44)
- [62] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) (ICSE '13). IEEE Press, Piscataway, NJ, USA, 772–781. <http://dl.acm.org/citation.cfm?id=2486788.2486890>
- [63] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.
- [64] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Propagation based local search for bit-precise reasoning. *Formal Methods in System Design* 51, 3 (2017), 608–636. <https://doi.org/10.1007/s10703-017-0295-6>
- [65] Aina Niemetz, Mathias Preiner, Armin Biere, and Andreas Fröhlich. 2015. Improving local search for bit-vector logics in SMT with path propagation. *Proceedings of DIFTS* (2015), 1–10.
- [66] Hristina Palikareva and Cristian Cadar. 2013. Multi-Solver Support in Symbolic Execution. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) (CAV 2013). Springer-Verlag, Berlin, Heidelberg, 53–68.
- [67] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) (ISSTA 2017). ACM, New York, NY, USA, 68–78. <https://doi.org/10.1145/3092703.3092728>
- [68] Corneliu Popescu and Wei-Ngan Chin. 2006. Inferring Disjunctive Postconditions. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues* (Tokyo, Japan) (ASIAN'06). Springer-Verlag, Berlin, Heidelberg, 331–345. <https://doi.org/10.5555/1782760>
- [69] Daniele Pretolani. 1996. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), 479–498.
- [70] Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Andres Nötzli, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2018. Rewrites for SMT solvers using syntax-guided enumeration. In *SMT Workshop*.
- [71] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling up DPLL (T) string solvers using context-dependent simplification. In *International Conference on Computer Aided Verification*. Springer, 453–474.
- [72] Lawrence Ryan. 2004. *Efficient algorithms for clause-learning SAT solvers*. Ph.D. Dissertation. Theses (School of Computing Science)/Simon Fraser University.
- [73] Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) (SAS'06). Springer-Verlag, Berlin, Heidelberg, 3–17. [https://doi.org/10.1007/11823230\\_2](https://doi.org/10.1007/11823230_2)
- [74] B. Schwarz, S. Debray, and G. Andrews. 2002. Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE 02)* (WCRA '02). IEEE Computer Society, USA, 45. <https://doi.org/10.5555/882506.885138>
- [75] Roberto Sebastiani. 2007. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3 (2007), 141–224.
- [76] Koushil Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) (ESEC/FSE-13). ACM, New York, NY, USA, 263–272. <https://doi.org/10.1145/1081706.1081750>
- [77] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) (PLDI 2018). ACM, New York, NY, USA, 693–706. <https://doi.org/10.1145/3192366.3192418>
- [78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
- [79] Ofer Shtrichman. 2000. Tuning SAT checkers for bounded model checking. In *International Conference on Computer Aided Verification* (CAV '02). Springer, 480–494.
- [80] João P. Marques Silva and Karem A. Sakallah. 1996. GRASP&MDash;a New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design* (San Jose, California, USA) (ICCAD '96). IEEE Computer Society, Washington, DC, USA, 220–227. <http://dl.acm.org/citation.cfm?id=244522.244560>
- [81] Rohit Singh and Armando Solar-Lezama. 2016. Swapper: A Framework for Automatic Generation of Formula Simplifiers Based on Conditional Rewrite Rules. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) (FMCAD '16). FMCAD Inc, Austin, TX, 185–192. <http://dl.acm.org/citation.cfm?id=3077629.3077661>
- [82] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.
- [83] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) (FSE '12). ACM, New York, NY, USA, Article 58, 11 pages. <https://doi.org/10.1145/2393596.2393665>
- [84] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. 2013. Overify: Optimizing Programs for Fast Verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.
- [85] Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. 2004. Refining the SAT Decision Ordering for Bounded Model Checking. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) (DAC '04). ACM, New York, NY, USA, 535–538. <https://doi.org/10.1145/996566.996713>
- [86] Yichen Xie and Alex Aiken. 2005. Scalable error detection using boolean satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) (POPL '05). ACM, New York, NY, USA, 351–363. <https://doi.org/10.1145/1040305.1040334>
- [87] Liangze Yin, Fei He, and Ming Gu. 2013. Optimizing the SAT Decision Ordering of Bounded Model Checking by Structural Information. In *Proceedings of the 2013 International Symposium on Theoretical Aspects of Software Engineering* (TASE '13). IEEE Computer Society, USA, 23–26. <https://doi.org/10.1109/TASE.2013.11>
- [88] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) (SEC'18). USENIX Association, Berkeley, CA, USA, 745–761. <http://dl.acm.org/citation.cfm?id=3277203.3277260>
- [89] Aleksandar Zeljić, Christoph M Wintersteiger, and Philipp Rümmer. 2016. Deciding bit-vector formulas with mcSAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 249–266.

# Relocatable Addressing Model for Symbolic Execution

David Trabish  
Tel Aviv University  
Israel  
davivtra@post.tau.ac.il

Noam Rinetzky  
Tel Aviv University  
Israel  
maon@cs.tau.ac.il

## ABSTRACT

Symbolic execution (SE) is a widely used program analysis technique. Existing SE engines model the memory space by associating memory objects with *concrete* addresses, where the representation of each allocated object is determined during its allocation. We present a novel addressing model where the underlying representation of an allocated object can be dynamically modified even after its allocation, by using *symbolic* addresses rather than concrete ones. We demonstrate the benefits of our model in two application scenarios: dynamic *inter-* and *intra*-object partitioning. In the former, we show how the recently proposed *segmented memory model* can be improved by dynamically merging several object representations into a single one, rather than doing that a-priori using static pointer analysis. In the latter, we show how the cost of solving array theory constraints can be reduced by splitting the representations of large objects into multiple smaller ones. Our preliminary results show that our approach can significantly improve the overall effectiveness of the symbolic exploration.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Symbolic execution, Addressing model, Memory partitioning

### ACM Reference Format:

David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397363>

## 1 INTRODUCTION

Symbolic execution (SE) is a widely used program analysis technique, that lies at the core of many applications including automatic test generation [3, 4, 27], bug finding [15], debugging [19], patch testing [23, 29], automatic program repair [24, 25], and reverse engineering [7]. During symbolic execution, the program is run with a *symbolic* input, rather than a concrete one. Whenever the

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397363>

symbolic executor reaches a branch that depends on a symbolic value, an SMT solver [12] is used to determine the feasibility of the branch sides, and the appropriate paths are further explored while updating their paths constraints with a corresponding constraint. A test case for a given explored path can be generated by asking the solver to generate a concrete assignment to the accumulated path constraints.

In modern symbolic executors such as KLEE [3] and ANGR [33], the address space of a symbolic state is modeled using a set of memory objects, where each memory object has a fixed concrete address and its own unique and non-overlapping address range. This model is a reasonable implementation choice, but identifying memory object addresses with concrete values is not essential: As long as the non-overlapping property of the memory objects holds, the values of the assigned addresses should not affect the execution.

We propose a new addressing model where the address expressions observable by the symbolic state are symbolic values rather than concrete ones. We preserve the non-overlapping property by maintaining additional *address constraints*, which constrain each symbolic address value to some constant value. This addressing model gives us the ability to *relocate* a given memory object, that is, modify its underlying address constraint in a way that is *transparent* to the symbolic state. Note that relocating a memory object is not possible under the existing addressing model, since an object is allocated with a fixed constant address that can't be modified.

To illustrate the benefits of such addressing model, consider the program from Figure 1, inspired by code found in our benchmarks. First, the program executes an initialization loop, where at each iteration it creates a hash table (line 43) and initializes it with some values (line 45). Then, at line 49 it performs a lookup on one of the tables with a symbolic key *k*. The lookup function `table_lookup` computes the hash of the input key, and iterates over the nodes of the relevant bucket to find the matching element. When the function `table_lookup` is called at line 49, the value of the pointer node at line 18 is symbolic, since it depends on the symbolic hash value which is derived from the symbolic value *k*. Therefore, at line 20, `node->key` dereferences a symbolic pointer.

Symbolic pointers present a challenge for SE [4, 20]. Modern SE systems associate each memory object with a different SMT array. Queries involving memory objects are then translated into SMT constraints involving the corresponding SMT arrays. When a symbolic pointer is dereferenced, the SE engine needs to *resolve* that symbolic pointer, that is, determine the memory objects it can refer to. If the symbolic pointer is resolved to more than one memory object, then we are in the case of *multiple resolution*. Several memory models for handling multiple resolutions have been considered in the past: The *forking model* [3, 27] forks the current symbolic state for each of the resolved objects, and in each forked state constrains the symbolic pointer expression to the range of the

resolved object. This approach is relatively efficient in terms of constraint solving, but may contribute to path explosion due to forking. The *merging model* [15, 33] creates a disjunction with one disjunct for each of the resolved objects. This way, forks are avoided but the path constraints become more complex due to the introduction of disjunctions.

Similarly to the merging model, the *segmented memory model* [20] proposes an approach which avoids forking, but achieves this by using *array theory* [14] rather than disjunction. In this model, the memory is split into segments using static pointer analysis [18, 30, 34, 35], such that each pointer (concrete or symbolic) refers to objects in a single segment. The segments are computed as follows: First, static pointer analysis is invoked to compute the points-to set of each pointer, that is, a set of abstract memory objects which are identified by static allocation sites. Then, every two intersecting points-to sets are merged into one points-to set, until a fixpoint is reached, that is all the points-to sets are disjoint. A segment is created for each of these disjoint points-to sets, such that all the memory objects associated with that points-to set will be allocated in that segment.

With this approach, forks are avoided when symbolic pointers are encountered, as each pointer is guaranteed to point to exactly one segment. However, this approach is limited by the precision of pointer analysis, and the created segments might contain too many objects. A big segment corresponds to a big SMT array<sup>1</sup>, which results in more complex constraints. To illustrate the limitations of pointer analysis, consider again the program at Figure 1. Pointer analysis can't distinguish between the different objects that are allocated at line 12, as they all have the same static allocation site. As a result, the bucket arrays of all 3 hash tables are allocated in one segment. Similarly, all the nodes allocated at line 33 are allocated in one segment as well. The SMT arrays of both segments are involved in the constraints, as both are accessed with a symbolic offset: The segment of buckets at line 18, and the segment of nodes at line 20. The sizes of these SMT arrays are at least three times bigger than those created by the forking model, due to merging of spurious objects. Therefore, despite of the reduction in the number of paths, the forking model still outperforms the segmented memory model in this case.

With our addressing model, we can dynamically relocate a memory object, in a way that is transparent to the symbolic state. Instead of determining the segments ahead of time, we create them on demand: If a symbolic pointer is resolved to multiple memory objects, we create a new segment and relocate the resolved memory objects to that segment. Now, a segment will contain only memory objects that can be pointed by a given symbolic pointer, without any spurious ones. The symbolic pointer at line 18 is resolved to one memory object, the bucket array of the first hash table, which doesn't require creating a segment. The symbolic pointer at line 20 is resolved only to nodes from the first hash table, so the created segment doesn't contain nodes from the other two hash tables. This way, we are able to avoid forking while creating smaller SMT arrays, which allows us to outperform both the segmented and the forking memory models.

<sup>1</sup>SMT arrays are actually unbounded, but the SE engine records the allocated size and never accesses elements beyond it.

Another challenge arising from the Program in Figure 1 relates to solving *array theory* [14] constraints. An array access with a concrete offset can be handled similarly to scalar variables, but accessing an array with a symbolic offset is more challenging, as the symbolic offset can refer to multiple locations in the array. In that case, the accessed value is expressed using an SMT formula over arrays, which creates a variable for each offset of the array. Such formulas are hard to solve, especially with big arrays, thus hindering symbolic execution. The hash tables created at line 43 are initialized with a bucket array of 300 entries, therefore the size of its corresponding SMT array is 2400 bytes. When calling `table_lookup` at line 49, the bucket is accessed with a symbolic offset at line 18, which triggers the usage of array theory. This constraint propagates into the path constraints that are created later during execution, thus slowing down the exploration.

With our addressing model, when a big enough memory object is accessed with a symbolic offset, we can relocate that memory object and split it into several smaller adjacent memory objects. For example, the symbolic pointer at line 18 was resolved to exactly one memory object, the bucket array of the hash table. Now, when this memory object is relocated and split, that symbolic pointer is resolved to multiple memory objects with smaller SMT arrays. In this case, despite of having more explored paths due to additional multiple resolutions, the reduced complexity of the constraints eventually results in faster exploration.

Our main contributions can be summarized as follows:

- (1) We propose a new addressing model, that allows *seamless* and *dynamic* relocation of memory objects during symbolic execution.
- (2) We provide an implementation based on KLEE [3], a state-of-the-art symbolic executor, which we make available as open source.<sup>2</sup>
- (3) We show the benefits of our addressing model in two scenarios: improving the segmented memory model, and reducing the cost of solving array theory constraints with big arrays.

## 2 PROPOSED ADDRESSING MODEL

In this section, we shortly describe the current addressing model of SE engines and present our relocatable model.

### 2.1 Existing Addressing Model

Symbolic executors, e.g., KLEE, represent the program's address space using a set of *memory objects*:

$$mo = (addr, size, arr) \in 2^{N^+ \times N^+ \times A}.$$

A memory object  $mo = (addr, size, arr)$  has a concrete *base address*  $addr \in N^+$  and spans  $size \in N^+$  bytes. In addition, a memory object is backed by an SMT array  $arr \in A$  with the same size  $size$ , which tracks the values stored into the memory object. If a value  $v$  is written to the  $e$ -th byte of the object, the array  $arr$  of  $mo$  is replaced by new array generated using a *store* expression:  $store(arr, e, v)$ . If

<sup>2</sup><https://www.tau.ac.il/~davivtra/projects/ram/>.

```

1 typedef struct node_s {
2     unsigned long data;
3     char *key;
4     struct node_s *next;
5 } node_t;
6 typedef struct {
7     node_t **buckets;
8     size_t size;
9 } table_t;
10
11 void table_init(table_t *t, size_t n) {
12     t->buckets = calloc(n, sizeof(node_t *));
13     t->size = n;
14 }
15
16 node_t *table_lookup(table_t *t, unsigned k) {
17     unsigned long h = hash(&k, sizeof(k)) % t->size;
18     node_t *node = t->buckets[h];
19     while (node != NULL) {
20         if (memcmp(&node->key, &k, sizeof(unsigned)) == 0) {
21             return node;
22         }
23         node = node->next;
24     }
25     return NULL;
26 }
27
28 void table_insert(table_t *t, unsigned k, int data) {
29     if (table_lookup(t, k)) {
30         return;
31     }
32     unsigned long h = hash(&k, sizeof(k)) % t->size;
33     node_t *node = calloc(1, sizeof(node_t));
34     node->key = k;
35     node->data = data;
36     node->next = t->buckets[h];
37     t->buckets[h] = node;
38 }
39
40 int main(int argc, char *argv[]) {
41     table_t tables[3];
42     for (unsigned i = 0; i < 3; i++) {
43         table_init(&tables[i], 300);
44         for (unsigned j = 0; j < 5; j++) {
45             table_insert(&tables[i], j, 7);
46         }
47     }
48     unsigned k; // symbolic
49     table_lookup(&tables[0], k);
50     return 0;
51 }

```

**Figure 1: Motivating example.**

the program accesses the  $e$ -th byte of the object, the read value is expressed using a *select* expression:  $\text{select}(arr, e)$ .<sup>3</sup>

The allocated objects span distinct addresses, i.e., for every two distinct memory objects  $(addr_1, size_1, arr_1)$  and  $(addr_2, size_2, arr_2)$  it holds that:

$$[addr_1, addr_1 + size_1] \cap [addr_2, addr_2 + size_2] = \emptyset.$$

<sup>3</sup> SE engines use an optimized representation of the memory object when the object is always accessed using concrete (non-symbolic) offsets. For simplicity, we avoid describing this optimization.

The non-overlapping requirement reflects the fact that different objects are located at different parts of the memory, and enables identifying memory objects by addresses: A concrete address  $addr$  can belong to at most one object. Thus, when the program accesses memory location  $addr$ , the SE engine can determine which SMT array represents the content at that address and act accordingly.

## 2.2 Relocatable Addressing Model

We propose a new addressing model, where the address of an object is a symbolic value, rather than a concrete one. As before, the program's address space is represented using a set of *memory objects*:

$$mo = (\alpha, size, arr) \in 2^{E \times N^+ \times A}.$$

However, the base address of an object is now a symbolic value  $\alpha \in E$  and not a concrete one. We enforce the non-overlapping property of different objects using *hidden* concrete base addresses: Whenever the program allocates a memory object, we create an *address pair* which consists of two values: a symbolic one  $\alpha$  and a concrete value  $c$ . The concrete value  $c$  is used to ensure that the allocated objects do not overlap in the same way that is done in the existing model. The symbolic value  $\alpha$  is the value that propagates to the symbolic state.

We maintain the correlation between the symbolic addresses and the concrete ones using *address constraints* (AC). These constraints, which are distinct from the path constraints of the symbolic state, record equalities of the form:

$$\alpha = e$$

where  $e$  is an expression over the hidden concrete addresses and the symbolic ones.

Note that the address constraints are not part of the path constraints, they are only used when we construct a query for the solver. Under this model, the address expressions stored in the symbolic state are symbolic, and any other expression might depend on these symbolic values, which are not constrained under the path constraints. Therefore, we substitute the address constraints before passing an expression  $e$  to the solver, that is:  $e[\alpha_i/e_i]$  (for each address constraint  $\alpha_i = e_i$ ).

*Remark.* Another way to achieve the non-overlapping property is to extend the path constraints of the symbolic state with appropriate constraints regarding the values of the base addresses. If we have memory objects  $(\alpha_1, size_1, arr_1), \dots, (\alpha_n, size_n, arr_n)$ , then we could have used the following constraints:

$$\forall i \neq j. \quad [\alpha_i, \alpha_i + size_i] \cap [\alpha_j, \alpha_j + size_j] = \emptyset.$$

However, we found this approach not scalable, as it adds additional constraints and symbolic values (depending on the number of objects in the symbolic state), making constraint solving significantly harder.

To illustrate our new addressing model, consider the program from Figure 2. When the array of pointers is allocated at line 2, we do the following: Assuming that a pointer size is 8 bytes, we first create a new memory object  $mo_1 = (\alpha_1, 16, arr_1)$  with an address pair  $(\alpha_1, c_1)$ , and add  $mo_1$  to the address space. Then, we add a new address constraint  $\alpha_1 = c_1$ , and the symbolic value  $\alpha_1$  is assigned to be the value of the local variable `array`. Note that  $c_1$  is chosen such that the addresses in the range  $[c_1, c_1 + 16]$  are distinct from those

```

1 #define N (2)
2 char **array = calloc(N, sizeof(char *));
3 for (unsigned int i = 0; i < N; i++)
4     array[i] = calloc(256, 1);
5
6 unsigned int i; // symbolic, i < 2
7 unsigned int j; // symbolic, j < 100
8 if (array[i][j] == 1)
9     // do something...

```

**Figure 2: A simple program allocating a two dimensional matrix using an array of pointers and multiple buffers.**

of any already allocated object. If the memory objects allocated at line 4 are  $mo_2$  and  $mo_3$  with the corresponding address pairs  $(\alpha_2, c_2)$  and  $(\alpha_3, c_3)$ , then before executing line 8 the address space consists of:

$$\{mo_1, mo_2, mo_3\}$$

and the address constraints are:

$$\{\alpha_1 = c_1, \alpha_2 = c_2, \alpha_3 = c_3\}$$

At line 8, where array is accessed with symbolic offset  $i$ , the value of  $a[i]$  is a *select* expression:

$$\text{select}(\text{store}(\text{store}(arr_1, 0, \alpha_2), 1, \alpha_3), i)$$

This symbolic pointer expression has to be resolved using the solver, but instead of passing the above expression we substitute our address constraints and the actual expression passed to the solver is:

$$\text{select}(\text{store}(\text{store}(arr_1, 0, c_2), 1, c_3), i)$$

With this model, objects can be now seamlessly relocated. Suppose that after the loop at line 5 we want to relocate the memory object  $mo_2$  to a new address. This is achieved by the following steps: First, we allocate a new memory object  $mo_4$  with an address pair  $(\alpha_4, c_4)$  of the same size as  $mo_2$ , and copy the contents of  $mo_2$  to  $mo_4$ . Second, we update the address space by removing  $mo_2$  and adding  $mo_4$ . Finally, we modify the address constraint  $\alpha_2 = c_2$  to be  $\alpha_2 = c_4$ . After the relocation, the expression obtained by the symbolic read  $a[i]$  at line 8 is the same as before:

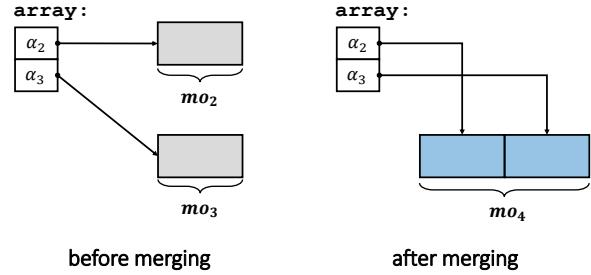
$$\text{select}(\text{store}(\text{store}(arr_1, 0, \alpha_2), 1, \alpha_3), i)$$

But now, the substituted expression that will be passed to the solver is different:

$$\text{select}(\text{store}(\text{store}(arr_1, 0, c_4), 1, c_3), i)$$

In this addressing model the address values observable by the symbolic state are always symbolic, but those passed to the solver are concrete, which allows efficient constraint solving. Using this addressing model we can perform *dynamically* two operations which are not possible with the existing model: merging multiple objects into one segment (Section 2.3), and splitting an object to multiple objects (Section 2.4).

*Limitations.* Our symbolic addressing model is applicable for *well-behaved* programs where the actual address of a memory object should not affect the behavior of the program. In particular, the program should only compare pointers of different objects for equality, as required by the C standard, and avoid using fixed addresses besides *null*.



**Figure 3: Merging multiple objects into a single segment.**

### 2.3 Application: Inter-object Partitioning

In this section, we show how the relocatable addressing model allows to dynamically merge the representations of several objects allocated by the program into a single memory object, thus reducing the cost of handling symbolic pointers.

**2.3.1 Segmented Memory Model.** Symbolic pointers present a particular challenge for SE [4, 20]. Since symbolic pointers can potentially refer to multiple memory objects, the SE engine first needs to *resolve* the pointer expression, that is, find all the memory objects to which the pointer could refer to, such that the right SMT arrays can be referenced. The case of multiple resolution, where we have more than one resolved object, is especially challenging, and several approaches have been proposed in the past.

In the *forking model* [3, 27], the current symbolic state is forked for each of the resolved objects, and each forked state constrains the symbolic pointer expression to the range of the resolved object. This approach is relatively efficient in terms of constraint solving, but may contribute to path explosion due to forking. The *merging model* [15, 33] uses disjunction to constrain the symbolic pointer to one of the resolved objects. This way, forks are avoided but the path constraints become more complex due to the introduction of disjunctions. Similarly to the merging model, the *segmented memory model* [20] also proposes an approach which avoids forking, but here this is achieved by using array theory rather than disjunction. In this model, the memory is split into segments, such that each symbolic pointer refers to objects in a single segment. The memory is partitioned into segments using conservative pointer analysis. The *points-to* set of a pointer is a set of abstract memory objects which are identified by their static allocation site (line). Two intersecting points-to sets are merged into one points-to set, until all the points-to sets are disjoint. For each of the (disjoint) points-to sets a new memory segment is created, such that all the memory objects associated with that points-to set will be allocated in that segment.

With this approach, objects pointed by any pointer (symbolic or concrete) are guaranteed to reside in exactly one segment, which makes the process of symbolic pointer resolution much more efficient. However, this approach is limited by the precision of pointer analysis, and the computed segments might be too large for complex programs. A larger segment results in a larger SMT array, thus affecting the performance of the solver.

**2.3.2 Dynamically Segmented Memory Model.** Instead of conservatively computing the segments ahead of time using pointer analysis, we propose a dynamic memory partitioning strategy that creates the segments on the fly using our relocatable addressing model. When we encounter a symbolic pointer that refers to multiple memory objects  $mo_1, \dots, mo_n$  where:

$$mo_i = (\alpha_i, size_i, arr_i)$$

we create a new segment (a memory object of an appropriate size), and relocate all these objects to this segment. First, we allocate a new segment  $(\alpha_s, size_s, arr_s)$  with an address pair  $(\alpha_s, c_s)$ , such that  $size_s = \sum_i size_i$ . Then we copy the contents of the memory objects  $mo_1, \dots, mo_n$  into that segment, such that after the copy it holds that:

$$\forall 0 \leq j < size_i. \quad select(arr_s, o_i + j) = select(arr_i, j)$$

$$\text{where } o_i = \sum_{k < i} size_k$$

Finally, we remove from the address space the memory objects  $mo_1, \dots, mo_n$ , and the address constraints on  $\alpha_i$  are updated to:

$$\alpha_i = \alpha_s + o_i$$

In the example from Figure 2, the symbolic pointer obtained by reading  $a[i]$  (at line 8) is resolved to two memory objects:  $mo_2 = (\alpha_2, 256, arr_2)$  and  $mo_3 = (\alpha_3, 256, arr_3)$ . We then create a new segment  $mo_4 = (\alpha_4, 512, arr_4)$  with the address pair  $(\alpha_4, c_4)$ , add  $mo_4$  to the address space, and add the address constraint  $\alpha_4 = c_4$ . Then we remove from the address space  $mo_2$  and  $mo_3$ , and update the address constraints of  $\alpha_2$  and  $\alpha_3$  to:

$$\alpha_2 = \alpha_4, \quad \alpha_3 = \alpha_4 + 256$$

After this transformation, our symbolic pointer is resolved to only one object ( $mo_4$ ), thus reducing the number of forks. Note that with this approach, the segments that we dynamically create don't contain redundant objects, those that are not pointed by the symbolic pointer. We merge only the objects that were resolved using the solver, thus reducing the size of the created segments.

The above transformation is graphically depicted in Figure 3. The state of the memory is shown before and after the merge transformation. Note that the contents of the object array are not affected by this transformation.

**2.3.3 Optimizations.** The segments created using our approach are guaranteed to be smaller, compared to the segments computed by pointer analyses based partition. However, the resolution process with our approach is more expensive, as a symbolic pointer still may point to multiple objects, before those are merged into one segment. The array theory constraints which are added due to the merging of objects are harder to solve, which makes constraint solving and symbolic pointer resolution more expensive. To address these issues, we propose several optimizations.

*Context Based Resolution.* Resolving symbolic pointers is a challenging task, as a symbolic pointer may refer to multiple memory objects. To determine these memory objects, symbolic executors (such as KLEE) scan the entire memory, and check for each scanned memory object an appropriate range condition using the solver. The dependence on the solver makes this process expensive, especially when the number of memory objects is high.

We observed that the resolution process can be optimized when using a context abstraction of the allocated objects. When a memory object is allocated, its  $k$ -context abstraction is obtained by the calling instructions of the last  $k$  stack frames, including the current instruction. Once we learn the contexts of the resolved objects at a given location (instruction), we can use that information to speed up the resolution process at the next time we have a resolution at that location. When objects are scanned during the resolution process, an object whose context is not one of the recorded contexts will be skipped, that is, a query will not be sent to the solver. Once the resolved objects are merged into a new segment (as described in Section 2.3.2), we can check the completeness of the resolution process with a single solver query that checks if the symbolic pointer must point to the newly created segment. If that's not the case, we fallback to the default resolution mechanism. Note that applying context-based resolution in the forking model is not beneficial, as checking completeness requires scanning the entire memory.

*Reusing Segments.* Modern symbolic executors use various heuristics for optimizing constraint solving. One of the key heuristics used in KLEE is query caching [3, 4], which associates the query expression to the result of the query. Consider again the program from Figure 2, and suppose that two symbolic states execute the branch instruction at line 8. With the dynamically segmented memory model, when the first state executes the branch, the resolved memory objects pointed by  $a[i][j]$  are merged to a new segment. Suppose that the memory object allocated at line 2 is  $(\alpha_1, size_1, arr_1)$ , and the created segment is  $(\alpha_2, 512, arr_2)$  with the address pair  $(\alpha_2, c_2)$ . In that case, the expression corresponding to the branch condition  $a[i][j] == 1$  after substituting the address constraints will be:

$$select(arr_2, (select(arr_1, i) + j) - c_2) = 1$$

Similarly, if in the second symbolic state the created segment is  $(\alpha_3, 512, arr_3)$ , then the expression for the same condition will be:

$$select(arr_3, (select(arr_1, i) + j) - c_3) = 1$$

The query caching is performed syntactically on the expression level, therefore the second symbolic state will have a cache miss for this query and will invoke the solver.

To handle this issue we attempt to reuse previously allocated segments. If at a program location  $L$ , a symbolic pointer was resolved to memory objects  $\{mo_i\}$  that were merged to a segment  $(\alpha_s, size_s, arr_s)$  whose address pair is  $(\alpha_s, c_s)$ , then we record the mapping between the tuples  $(L, \{mo_i\})$  and  $(c_s, arr_s)$ . If later another symbolic state resolves a symbolic pointer at program location  $L$  to the same set of memory objects  $\{mo_i\}$ , the created segment will be  $(\alpha'_s, size_s, arr_s)$  with the address pair  $(\alpha'_s, c_s)$ .

This way, when the second symbolic state performs the merge at line 8, its address pair will be  $(\alpha_3, c_2)$ , and its SMT array will be  $arr_2$ . Therefore, the expression of the branch condition will be equal to that of the first symbolic state, which will result in a cache hit.

## 2.4 Application: Intra-object Partitioning

In this section, we show how the relocatable addressing model can dynamically transform the memory state such that a single object allocated by the program can be represented by several adjacent

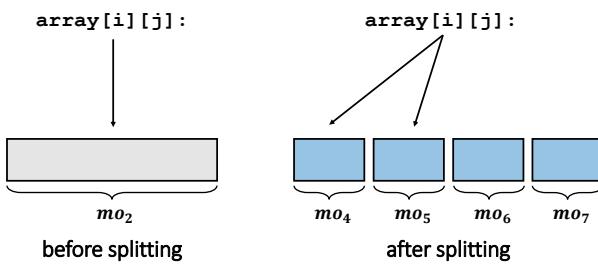


Figure 4: Splitting an object into adjacent smaller objects.

smaller memory objects, thus reducing the size of the SMT arrays and consequently the cost of constraint solving.

When a memory object is accessed with a symbolic offset, the resulting values are translated using array theory to *select* and *store* expressions. Solving array theory constraints is usually much harder than regular bit-vector constraints, especially when the arrays are big. During symbolic execution, we might have many such queries, which then results in a significant slowdown.

To make constraint solving more efficient, we attempt to reduce the size of big SMT arrays by dynamically splitting their corresponding memory objects into smaller ones. We split only memory objects which were accessed with a symbolic offset, as array theory will not be used for memory objects that are accessed only with concrete offsets.

The split operation on a memory object  $mo = (\alpha, size, arr)$  with an address pair  $(\alpha, c)$  works as follows: We allocate  $n$  new memory objects  $mo_1, \dots, mo_n$  with address pairs  $(\alpha_i, c_i)$ , such that their addresses are consecutive:

$$\forall 1 \leq i < n. c_{i+1} = c_i + size_i$$

and add the address constraints  $\{\alpha_i = c_i\}$ . We initialize the contents of each memory object  $mo_i = (\alpha_i, size_i, arr_i)$  using  $mo$  such that:

$$\forall 0 \leq j < size_i. select(arr_i, j) = select(arr, o_i + j)$$

$$\text{where } o_i = \sum_{k < i} size_k$$

Then we remove from the address space the memory object  $mo$ , and update the address constraint on  $\alpha$  to  $\alpha = \alpha_1$ , the symbolic address of the first split memory object. The sizes  $\{size_i\}$  of the memory objects  $\{mo_i\}$  are determined according to a given partitioning strategy.

Consider the symbolic execution of the program from Figure 2 under the forking model. Assuming that the memory object allocated at line 2 is  $(\alpha_1, 16, arr_1)$ , the expression of the pointer from which the value  $a[i][j]$  is read is:

$$select(arr_1, i) + j$$

This symbolic pointer is resolved to the two objects allocated at line 4, namely  $mo_2$  and  $mo_3$ , and the symbolic state is forked. If we continue the execution with the symbolic state which constrains the symbolic pointer to the memory object  $mo_2 = (\alpha_2, 256, arr_2)$ , as depicted at the upper part of Figure 4, then the value of  $a[i][j]$

would be:

$$select(arr_2, select(arr_1, i) + j - \alpha_2)$$

Since this value is read from  $mo_2$  with a symbolic offset, we would like to split  $mo_2$ . Suppose that we choose a partitioning strategy that splits a given memory object into  $n$  memory objects of equal size. Then for  $n = 4$  we split  $mo_2$  into 4 objects:  $mo_4, mo_5, mo_6, mo_7$ , as depicted in the lower part of Figure 4. After the split, we re-execute the load instruction that references the previous symbolic pointer, but now it is resolved to two objects,  $mo_4$  and  $mo_5$ , due to the constraint  $j < 100$ .

Assuming that  $mo_4 = (\alpha_4, size_4, arr_4)$ , in the first newly forked state the expression of the value  $a[i][j]$  is now:

$$select(arr_4, select(arr_1, i) + j - \alpha_4)$$

Due to the split we explore an additional path, the one that constrains the symbolic pointer to  $mo_5$ , but we get smaller SMT arrays:  $arr_4$  and  $arr_5$ . The size of  $arr_4$  is 64, which is four times smaller than the size of  $arr_2$ , which makes the new expression much easier to solve.

The effect of the split operation depends on the partitioning strategy: For smaller split objects the SMT arrays are smaller, but the number of resolved objects is higher, and therefore the number of forks is higher as well. For bigger split objects the number of resolved objects and forks is lower, but the resulting SMT arrays are consequently bigger. The number of split objects also affects the resolution process which works by scanning the entire memory. We investigate this trade-off in Section 4.3.

### 3 IMPLEMENTATION

We implemented our addressing model on top of the KLEE symbolic execution engine [3], configured with LLVM 7.0.0 and STP 2.3.3. We modified KLEE’s allocation API to return symbolic addresses instead of concrete ones, and extended the symbolic state with the address constraints. Our addressing model is actually implemented as a mixed *concrete-symbolic* one, that is, we allow allocation of memory objects with concrete addresses as well. Obviously, the applications described in Sections 2.3 and 2.4 can’t be applied to such memory objects.

In our implementation, symbolic addresses can be assigned to both stack and heap memory objects. By default, we don’t create symbolic addresses for stack variables, as they are rarely involved in multiple resolutions or array theory constraints. From technical reasons we currently don’t support global variables with symbolic addresses, but this can be solved by automatically rewriting the program such that global variables would be allocated on the heap upon the program’s startup.

When a memory object is split, we need to ensure that reads and writes to primitive fields are performed within a single memory object. We assume that struct fields are aligned to 8 bytes, so the size of each split object must be aligned to 8 bytes as well.

### 4 EVALUATION

We perform several experiments in our evaluation: In Section 4.1, we show the correctness of our addressing model. In Sections 4.2 and 4.3 respectively, we show the benefits of our addressing model

**Table 1: The benchmarks used throughout the evaluation, with their versions and number of source code lines (SLOC).**

Benchmark	Version	SLOC
<i>m4</i>	1.4.18	80K
<i>make</i>	4.2	28K
<i>sqlite</i>	3.21	127K
<i>apr</i>	1.6.3	60K
<i>gas</i>	2.31.1	266K
<i>libxml2</i>	2.9.8	197K
<i>coreutils</i>	8.31	188K

when applied in the context of inter-object partitioning (dynamic merging) and intra-object partitioning (dynamic splitting).

*Experimental Setup.* We performed our experiments on an a machine running Ubuntu 16.04, equipped with an Intel i7-6700 processor (8 cores) and 32GB of RAM.

*Benchmarks.* The benchmarks used in our evaluation are listed in Table 1. *GNU m4*<sup>4</sup> is a macro processor included in most Unix-based systems. *GNU Make*<sup>5</sup> is a tool which controls the generation of executables and other non-source files, also widely used in Unix-based systems. *SQLite*<sup>6</sup> is one of the most popular SQL database libraries in the world. *Apache Portable Runtime*<sup>7</sup>, (APR) is a library used by the Apache HTTP server that provides cross-platform functionality for memory allocation, file operations, containers, and networking. *GNU Assembler*<sup>8</sup>, commonly known as *gas*, is the assembler used by the GNU project, and is the default back-end of GCC. The *libxml2*<sup>9</sup> library is a XML parser and toolkit developed for the Gnome project. *GNU Coreutils*<sup>10</sup> is a collection of utilities for file, text, and shell manipulation.

## 4.1 Correctness

In this experiment, we empirically validate the correctness of our addressing model. To do so, we check that the existing addressing model (vanilla KLEE) and our model are consistent in terms of path exploration. Here we use our addressing model without applying the merging (Section 2.3) and splitting (Section 2.4) operations, therefore the number of explored paths in both models is expected to be identical. For this experiment, we used the programs listed in Table 1, where in *coreutils* we selected 15 programs which behave deterministically across multiple runs.

For each program, we proceed with the following evaluation process: First, we run KLEE for roughly one hour, and record the number of executed instructions. Then, we run KLEE again up to the number of recorded instructions, with both its default addressing model and our addressing model. Finally, we validate that the

<sup>4</sup><https://www.gnu.org/software/m4/>

<sup>5</sup><https://www.gnu.org/software/make/>

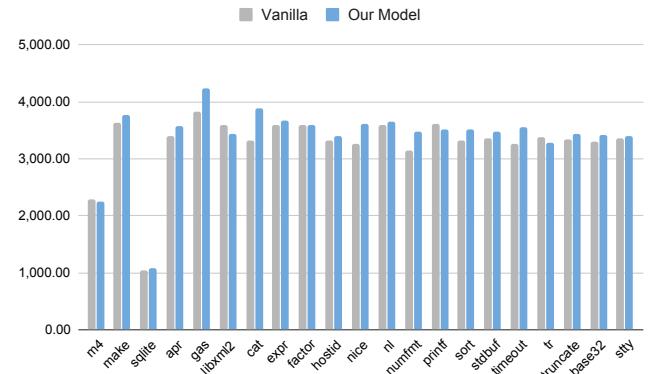
<sup>6</sup><https://www.sqlite.org/>

<sup>7</sup><https://apr.apache.org/>

<sup>8</sup><https://sourcery.mentor.com/binutils/docs/as/>

<sup>9</sup><http://www.xmlsoft.org/>

<sup>10</sup><https://www.gnu.org/software/coreutils/>



**Figure 5: The termination times in seconds for each program with the existing addressing model (Vanilla) and our addressing model (Our Model).**

number of paths explored by both addressing models is identical. To enforce determinism, we ran each program with the DFS search heuristic, using the deterministic memory allocator.

Our experiments confirmed that the number of explored paths with both addressing models is indeed the same. We also measured the runtime overhead induced by our addressing model, which comes from substituting expressions and maintaining additional symbolic values for address expressions, as described in Section 2.2. Figure 5 shows the termination time (in seconds) for each program with the two addressing models. For the programs we tested, the maximum runtime overhead was 16% (in *cat* from *coreutils*), and the average runtime overhead was 4%.

## 4.2 Inter-object Partitioning

**4.2.1 Dynamically Segmented Memory Model.** In this experiment we compare the performance of our dynamically segmented memory model (DSMM) with the segmented memory model (SMM) proposed in [20], and the forking model (FMM) used in vanilla KLEE. We perform the same experiment as in [20] (Figures 6,7,8,9 from the original paper). The benchmarks of this experiment are: *m4*, *make*, *sqlite*, and *apr*<sup>11</sup>. These programs create symbolic pointers which trigger multiple resolutions, as they all use hash tables with symbolic keys. Programs in which symbolic pointers with multiple resolutions are few (*gas*) or absent (*libxml2*, and *coreutils*) are not used in this experiment. The impact of our addressing model on the runtime overhead for these programs is discussed in Section 4.1.

We run each program with a timeout of 24 hours using three search heuristics (DFS, BFS and KLEE’s default search heuristic) with the deterministic memory allocator, and measure the termination time and the memory consumption with all three memory models. Our dynamically segmented memory model is run with several optimizations, whose impact is investigated in Section 4.2.2.

<sup>11</sup> In *sqlite* we disabled the *counter-example caching* query optimization, as it lead to inconsistent termination times. For SMM, the result was timeout with and without this optimization. Note that this optimization is different from the *query caching* discussed in Section 2.3.3.

**Table 2: Maximum segment size in bytes.**

Program	Max. Size	
	SMM	DSMM
<i>m4</i>	2753	1008
<i>make</i>	7574	1776
<i>sqlite</i>	17604	528
<i>apr</i>	8316	240

**Table 3: Termination time in hh:mm or TO (timeout) and memory usage in GB or OOM (out-of-memory) with different memory models: FMM, SMM and DSMM.**

	Search	Time			Memory		
		FMM	SMM	DSMM	FMM	SMM	DSMM
<i>m4</i>	DFS	21:03	01:04	00:26	0.1	0.2	0.3
	BFS	09:32	01:04	00:30	OOM	0.3	0.4
	Default	09:41	01:10	00:35	OOM	0.5	0.5
<i>make</i>	DFS	06:35	22:49	03:35	1.6	0.8	0.5
	BFS	06:33	23:03	03:34	1.6	0.8	0.6
	Default	07:27	23:04	03:38	1.5	0.8	0.4
<i>sqlite</i>	DFS	00:18	TO	01:36	0.2	0.8	0.4
	BFS	00:18	TO	01:34	0.3	0.7	0.4
	Default	00:18	TO	01:36	0.2	0.7	0.5
<i>apr</i>	DFS	01:01	00:07	00:19	0.1	0.1	0.1
	BFS	00:59	00:07	00:19	0.1	0.1	0.1
	Default	00:57	00:07	00:19	0.1	0.1	0.1

Table 2 shows the maximum segment sizes for both SMM and DSMM. The maximum segment size created using our approach is reduced on average by 83%, where the reduction is 63% in *m4*, 77% in *make*, 97% in *sqlite*, and 97% in *apr*. The sizes of the created segments are crucial for the performance, as the sizes of their corresponding SMT arrays affect the complexity of constraint solving.

Table 3 shows for each benchmark and search heuristic the termination time and the memory consumption obtained with the three memory models. We first discuss the performance comparison between SMM and DSMM, and then discuss the performance of FMM compared to the segmentation-based memory models (SMM and DSMM).

In *m4*, our approach achieves an average speedup of 2.2× compared to the segmented memory model, with a slightly higher memory usage. In *make*, our approach achieves an average speedup of 6.3× compared to the segmented memory model, and the memory usage is roughly the same. In *sqlite*, the segmented memory model doesn't terminate before the 24 hours time limit (for all search heuristics), which results in an average speedup of at least 14.2× for our approach. The memory usage is roughly the same with both approaches in this case. In *apr*, the memory usage is equally low in both approaches, but in terms of termination time, this is the only case where our approach performs worse. As was mentioned

above, the maximum segment size with our approach is significantly smaller, but the segmented memory model is still 2.3× faster on average. The program allocates several small objects using *libc*'s standard allocation API, and some other objects using a custom pool allocator, that internally uses an array of 8192 bytes. During the symbolic execution of the program, the SMT array associated with the big array (of the pool allocator) is involved in the queries, which slows down the exploration. In the case of the segmented memory model, the big array and the other small objects are merged into one segment. With our approach, some of the small objects are dynamically merged into one segment, but the big array remains untouched. In both approaches, we have a big array of roughly the same size which is involved in the constraints, thus slowing down the solver. The advantage of our approach is having smaller segments, but symbolic pointers are still needed to be resolved (possibly to multiple memory objects), which leads to higher resolution time. In the segmented memory model a symbolic pointer is guaranteed to point to one segment at most, so the resolution process is less costly. In this case, the resolution process with our approach is indeed higher, as each resolution query involves the big array that was mentioned before. Actually, the resolution process takes roughly 50% of the total execution time, which explains the extra time required for our approach to terminate.

When comparing the performance of FMM with the segmentation-based memory models (SMM and DSMM), the results are mixed. In *m4* and *apr*, FMM performs significantly slower with all search heuristics than other memory models. The memory usage in *apr* is basically identical across all memory models, but in *m4* the memory usage with FMM reaches the limit (with BFS and Default), which leads to an incomplete exploration and early termination. In *make*, FMM performs faster than SMM but slower than DSMM, and its memory usage is higher compared to other memory models. In *sqlite*, FMM outperforms SMM and DSMM in terms of termination time and memory usage.

**4.2.2 Optimizations.** Here we further investigate the impact of the optimizations discussed in Section 2.3.3. We use the same benchmarks as in Section 4.2.1, and run each of them with the DFS search heuristic, using the deterministic memory allocator.

**Context-Based Resolution.** To understand how a given context abstraction affects the process of symbolic pointer resolution, we examine the number of resolution queries (those which are created during a resolution). We evaluate the default resolution mechanism which scans the entire memory, and our context-based resolution with the  $k$ -context abstraction which takes into account the last  $k$  calling instructions from the stack trace of a given allocation site, including the current instruction.

Table 4 shows the number of resolution queries in different modes: default resolution and context-based resolution with  $0 \leq k \leq 4$ . The largest reduction occurs when  $k = 4$ , where the number of queries is decreased by 44% in *m4*, 71% in *make*, 82% in *sqlite*, and 2% in *apr*. We can also see that as  $k$  increases, the number of resolution queries (non-strictly) decreases. The impact of  $k$  on the reduction rate varies across benchmarks: In *make* and *m4* we have a significant reduction for  $k = 0, 1$ , but increasing  $k$  further does not result in a significant improvement. In *sqlite* a reduction of 25% is obtained already with  $k = 0$ , and increasing  $k$  further

**Table 4: The number of resolution queries with different context abstractions.**

Program	Default	K-Context				
		0	1	2	3	4
<i>m4</i>	12050	11434	6920	6920	6808	6808
<i>make</i>	8104	2632	2365	2365	2365	2365
<i>sqlite</i>	9134	6816	6816	3320	3320	1668
<i>apr</i>	96	94	94	94	94	94

to  $k = 2, 4$  results in a reduction of 64% and 82%, respectively. Compared to other benchmarks, the number of created memory objects in *apr* is relatively small, so the resolution process with the default mechanism is almost optimal. The number of resolution queries is reduced by only two queries for  $k = 0$ , and using higher values does not give any better results.

Columns None and  $\text{Opt}_1$  of Table 5 show the termination time with the default resolution mechanism and with context-based resolution (for  $k = 4$ ). In *m4*, *make*, and *sqlite*, the termination time was reduced by 26%, 13%, and 62% respectively. In *apr*, the number of reduced resolution queries was minor, therefore the termination time was not affected. Note that the reduction in termination time depends not only on the number of reduced queries, but also on the relative proportion of resolution time: If the resolution time is already low, then reducing the number of resolution queries is likely to result in a minor improvement. When the resolution time is high, a significant speedup can be achieved even with a minor reduction in the number of resolution queries.

In general, note that using the highest value for  $k$  (or a full-context abstraction with  $k = \infty$ ) is not guaranteed to be beneficial: If the value of  $k$  is too high, our context-based resolution might skip too many memory objects, which will result in an *incomplete* resolution.

*Reusing Segments.* The reusing segments optimization attempts to achieve speedup by improving the solver query caching, that is, reducing the number of solver queries which are actually passed to the SMT solver.

The impact of reusing segments can be seen in columns None and  $\text{Opt}_2$  of Table 5. In *m4* and *apr*, the termination time was reduced by 85% and 17% respectively, while in *make* and *sqlite* the reduction was relatively small with 3% and 8% respectively. As was mentioned before, the benchmarks in this experiment use hash tables with buckets, which are typically implemented using an array of pointers. In the case of *m4* and *apr*, the hash tables are initialized at startup, and are not modified after that. Therefore, an SMT array that corresponds to an array of pointers is identical for all the symbolic states, which allows efficient caching when segments are reused. In the case of *make* and *sqlite*, the hash tables are modified after the initialization, so when different states update a hash table by adding a new element, the corresponding SMT arrays are different as well, which makes the reuse mechanism less efficient. To mitigate these issues, one can try to reuse addresses for memory objects in general, not only segments. We leave this direction for future research.

**Table 5: Termination time in hh:mm in different modes:**  
**None:** without any optimizations,  **$\text{Opt}_1$ :** with context-based resolution (for  $k = 4$ ),  **$\text{Opt}_2$ :** with reusing segments, and **Both:** with both optimizations.

Program	Termination Time			
	None	$\text{Opt}_1$	$\text{Opt}_2$	Both
<i>m4</i>	03:34	02:37	00:33	00:26
<i>make</i>	04:14	03:42	04:06	03:35
<i>sqlite</i>	04:16	01:37	03:58	01:36
<i>apr</i>	00:23	00:23	00:19	00:19

**Table 6: Maximum size of split objects.**

Program	Size
<i>m4</i>	4072
<i>make</i>	8192
<i>sqlite</i>	328
<i>apr</i>	8192
<i>gas</i>	524411
<i>libxml2</i>	4096

### 4.3 Intra-object Partitioning

In this experiment, we investigate the impact of splitting arrays on the termination time and the number of explored paths, in programs that create array theory constraints with big arrays. For each program we compare the results obtained by vanilla KLEE and the splitting approach with different partitioning strategies. When the splitting approach is used with a partitioning strategy  $P_n$ , a memory object is split into smaller memory objects of size  $n$ . We use a *split threshold* of 300 bytes, that is, we split only memory objects whose size is bigger than that given threshold. We use the DFS search heuristic and the deterministic memory allocator.

The benchmarks in this experiment are: *m4*, *make*, *sqlite*, *apr*, *gas*, and *libxml2*. Similarly to the experiments in Section 4.2, we achieve termination by running these programs with a partially symbolic input, except for *libxml2* which is run with a fully symbolic input. In Section 4.2, the programs *m4* and *make* (which were taken from [20]) were run with decreased sizes for some of the arrays: In *m4*, the hash table size was decreased using one of the program’s command line flags ( $-H$ ), and in *make* some of the arrays were manually patched to have smaller sizes. In this experiment we restore the default array sizes, in order to test the splitting approach with arrays which are big enough.

Table 7 shows the termination time and the number of paths with both vanilla KLEE and our splitting approach (with different partitioning strategies). In terms of termination time, the speedup of the splitting approach relative to vanilla KLEE varies between  $6.0\times$ - $13.4\times$  in *m4*,  $1.2\times$ - $17.9\times$  in *make*,  $0.9\times$ - $4.0\times$  in *sqlite*,  $13.3\times$ - $132.1\times$  in *apr*,  $33.6\times$ - $43.8\times$  in *gas*, and  $1.0\times$ - $2.5\times$  in *libxml2*. Nevertheless, there were two cases where our approach performed worse: In

*sqlite*, the size of the split object is 328 bytes, therefore using  $P_{512}$  affects neither the memory objects nor the number of explored paths, with the termination time being higher by 4% due to the overhead incurred by our addressing model. Running *libxml2* with  $P_{32}$  resulted in a slightly higher termination time, mainly due to the increased number of explored paths.

Table 6 shows the maximum size of a split object for each benchmark. The sizes vary between roughly 500KB in *gas* and only 328 bytes in *sqlite*, which shows that the splitting approach can be successfully applied with both big arrays and relatively small ones.

The partitioning strategy used in the splitting approach directly affects the termination time and the number of explored paths. When an object is split according to some partitioning strategy, a more refined partition will (non-strictly) increase the number of memory objects that a symbolic pointer can point to. Since we are in the forking model, when we decrease  $n$ , that is refine the partition, the number of resolved memory objects with  $P_n$  increases together with the number of explored paths. In addition, when the partitioning is more refined, the number of memory objects grows, which may result in a slower symbolic pointer resolution. Nevertheless, using a more refined partitioning creates smaller SMT arrays, which makes constraint solving easier. This tradeoff between the complexity of the constraints on one side, and the number of paths and the resolution time on the other, eventually determines the termination time with a given partitioning strategy.

When trying to understand the impact of a given partitioning strategy, we observed two main patterns. When  $n$  is decreased in *sqlite* and *apr*, the overhead of forks and resolution remains relatively low and SMT arrays also become smaller, which results in better overall performance. In other benchmarks (*m4*, *make*, *gas*, and *libxml2*), decreasing  $n$  toward small values (32) results in a slowdown due to an increased number of explored paths. In *make* and *m4*, increasing  $n$  too much toward high values (512) results in a slowdown as well, due to the growing complexity of constraints over bigger SMT arrays. The sweet spot value for  $n$  lies somewhere between 64 and 256.

## 5 RELATED WORK

Coppa et al. [10] model the symbolic memory as a set of tuples, where each tuple associates an address expression to a value expression, along with a timestamp, and a condition. When a write is performed, the memory is updated with a new tuple containing the corresponding address and value expressions. When a read is performed, the memory is scanned to determine the tuples that match the given address expression. The read value is then expressed using an *if-then-else* expression, which is built using the matching tuples. This approach attempts to improve the merging memory model used in ANGR [33], by avoiding concretizations of symbolic pointers when they are encountered in reads or writes. In our work, accessing memory objects with symbolic offsets is handled using array theory. The segmented memory model, in its static [20] and dynamic forms, is similar in spirit to the merging approach, but here instead of using *if-then-else* expressions or disjunctions, we compute some memory partitioning while using array theory. Our splitting approach attempts to improve the constraint solving of array theory constraints which are not used in [10].

**Table 7: Termination time in *hh:mm:ss* and number of explored path with vanilla KLEE and different splitting strategies.**

Program	Mode	Time	Paths
<i>m4</i>	Vanilla	00:39:46	82
	$P_{512}$	00:06:40	82
	$P_{256}$	00:02:58	101
	$P_{128}$	00:03:16	145
	$P_{64}$	00:03:50	257
	$P_{32}$	00:05:08	485
<i>make</i>	Vanilla	09:04:13	3386
	$P_{512}$	01:13:50	4488
	$P_{256}$	00:30:26	6088
	$P_{128}$	00:39:47	10136
	$P_{64}$	01:53:15	21304
	$P_{32}$	07:44:19 <sup>12</sup>	55928
<i>sqlite</i>	Vanilla	00:18:38	147
	$P_{512}$	00:19:26	147
	$P_{256}$	00:15:21	213
	$P_{128}$	00:09:46	259
	$P_{64}$	00:07:26	355
	$P_{32}$	00:04:38	465
<i>apr</i>	Vanilla	01:01:38	961
	$P_{512}$	00:04:39	1024
	$P_{256}$	00:02:21	1225
	$P_{128}$	00:01:16	1444
	$P_{64}$	00:00:47	1849
	$P_{32}$	00:00:28	2025
<i>gas</i>	Vanilla	05:18:26	5
	$P_{512}$	00:07:16	5
	$P_{256}$	00:07:25	5
	$P_{128}$	00:07:43	5
	$P_{64}$	00:08:23	5
	$P_{32}$	00:09:29	5
<i>libxml2</i>	Vanilla	01:22:27	4413
	$P_{512}$	00:33:26	5003
	$P_{256}$	00:33:38	5230
	$P_{128}$	00:39:13	5821
	$P_{64}$	00:59:06	6885
	$P_{32}$	01:23:00	8718

The idea of modeling addresses *not* as constant values was discussed in past work. For example, Hajdu et al. [17] model address values in smart contracts as un-interpreted symbols as in this context addresses can be only queried for equivalence. We allow for arbitrary queries over symbolic addresses.

The idea of using memory partitioning for improving program analysis has been explored before. In the context of bounded model

checking, partitioned models have been used based on various complementary analyses such as points-to analysis [1, 36, 37], data structure analysis (DSA) [21], and type based analysis [2, 9]. CBMC [8] and ESBMC [11] also use points-to analysis to refine their memory models. SeaHorn [16] uses a context-insensitive variant of DSA [21]. The memory partitioning used in prior work is computed ahead of time, while our dynamically segmented memory model doesn't require additional pre-computations, and enables a *path-specific* memory partitioning during runtime, thus resulting in a more accurate partitioning.

Our context-based resolution uses the last  $k$  call sites to learn the contexts of resolved objects in order to accelerate the process of symbolic pointer resolution. The usage of call-site abstraction is inspired by context-sensitivity in program analysis [32].

Scaling constraint solving is a well known challenge in symbolic execution [5, 6]. Prior work has used different optimizations such as arithmetic transformations [4, 31], caching query results [3, 4], caching counter examples [3, 38, 39], splitting constraints into independent sets [4], multiple solvers support [26], interval-based solving [13], and even fuzzing-based solving [22]. Perry et al [28] focus on reducing the cost of array theory constraints using several semantics-preserving transformations. These transformations attempt to eliminate array constraints as much as possible by replacing them with constraints over their indices and values. The approaches mentioned above are orthogonal to our splitting approach, with which they could be combined.

## 6 CONCLUSION AND FUTURE WORK

We presented a novel addressing model where the underlying representations of allocated objects can be dynamically modified, by using *symbolic* addresses rather than concrete ones. We showed how this model can improve the existing segmented memory model, and reduce the cost of solving array theory constraints.

We presented our addressing model, and its two applications, assuming the usage of array theory (as is the case in KLEE). In general, the merging approach discussed in Section 2.3 does not require using array theory, and regardless of the fact the one of the optimizations (2.3.3) is array theory specific, we believe that the core idea can be applied to other symbolic executors as well.

Our work opens the opportunity for different research directions: In our dynamic intra-object partitioning, we used a rather simple partitioning strategy. Automatically determining and customizing the partitioning strategy for a given object in a program might further improve performance. Another challenge is predicting when a splitting or merging transformation is likely to payoff. In addition, these approaches could be applied simultaneously.

## ACKNOWLEDGEMENTS

The research leading to these results has received funding from the Lev Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, Pazy Foundation, and Israel Science Foundation (ISF) grants No. 1996/18.

<sup>12</sup>In this configuration, we use the standard (libc) memory allocator and not the deterministic one used in all other experiments as the latter does not reuse addresses and ran out of space during the experiment.

## REFERENCES

- [1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Technical Report.
- [2] Rodney M Burstall. 1972. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence* 7, 23–50 (1972), 3.
- [3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, USA).
- [4] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)* (Alexandria, VA, USA).
- [5] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1066–1071.
- [6] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. <https://doi.org/10.1145/2408776.2408795>
- [7] Vitaly Chipounov and George Candea. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)* (Paris, France).
- [8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)* (Barcelona, Spain).
- [9] Jeremy Condit, Brian Hackett, Shuvendu K Lahiri, and Shaz Qadeer. 2009. Unifying type checking and property checking for low-level code. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 302–314.
- [10] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking pointer reasoning in symbolic execution. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 613–618.
- [11] L. Cordeiro, B. Fischer, and J. Marques-Silva. 2012. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering (TSE)* 38, 4 (July 2012), 957–974.
- [12] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
- [13] Oscar Soria Dustmann, Klaus Wehrle, and Cristian Cadar. 2018. PARTI: a multi-interval theory solver for symbolic execution.
- [14] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)* (Berlin, Germany).
- [15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)* (San Diego, CA, USA).
- [16] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.
- [17] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for Solidity smart contracts. *arXiv preprint arXiv:1907.04262* (2019).
- [18] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proc. of the 2nd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)* (Snowbird, UT, USA).
- [19] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)* (Zurich, Switzerland).
- [20] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia). ACM, 774–784.
- [21] Chris Lattner, Andrew Lenhardt, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'07)* (San Diego, CA, USA).
- [22] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 521–532.
- [23] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia).
- [24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. ACM, 691–701.

- [25] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)* (San Francisco, CA, USA).
- [26] Hristina Palikareva and Cristian Cadar. 2013. Multi-solver Support in Symbolic Execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)* (Saint Petersburg, Russia).
- [27] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehltz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. (Sept. 2013).
- [28] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating Array Constraints in Symbolic Execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)* (Santa Barbara, CA, USA).
- [29] David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. of the 24th USENIX Security Symposium (USENIX Security'15)* (Washington, D.C., USA).
- [30] Barbara G. Ryder. 2003. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Compiler Construction*, Görel Hedin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–137.
- [31] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (Lisbon, Portugal).
- [32] M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S.S. Muchnick and N.D. Jones (Eds). Prentice-Hall, Englewood Cliffs, NJ, Chapter 7.
- [33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'16)* (San Jose, CA, USA).
- [34] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. <https://doi.org/10.1561/2500000014>
- [35] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. *Alias Analysis for Object-Oriented Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 196–232.
- [36] Bjarne Steensgaard. 1996. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*. Springer, 136–150.
- [37] B. Steensgaard. 1996. Points-to analysis in almost-linear time. In *Principles of Programming Languages (POPL)*.
- [38] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)* (Cary, NC, USA).
- [39] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)* (Minneapolis, MN, USA).



# Running Symbolic Execution Forever

Frank Busse  
Imperial College London  
United Kingdom  
f.busse@imperial.ac.uk

Martin Nowack  
Imperial College London  
United Kingdom  
m.nowack@imperial.ac.uk

Cristian Cadar  
Imperial College London  
United Kingdom  
c.cadar@imperial.ac.uk

## ABSTRACT

When symbolic execution is used to analyse real-world applications, it often consumes all available memory in a relatively short amount of time, sometimes making it impossible to analyse an application for an extended period. In this paper, we present a technique that can record an ongoing symbolic execution analysis to disk and selectively restore paths of interest later, making it possible to run symbolic execution indefinitely.

To be successful, our approach addresses several essential research challenges related to detecting divergences on re-execution, storing long-running executions efficiently, changing search heuristics during re-execution, and providing a global view of the stored execution. Our extensive evaluation of 93 Linux applications shows that our approach is practical, enabling these applications to run for days while continuing to explore new execution paths.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

symbolic execution, memoization, KLEE

### ACM Reference Format:

Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397360>

## 1 INTRODUCTION

For testing real-world software systems, symbolic execution is often proposed as a method for thoroughly enumerating and testing every potential path through an application. While achieving full enumeration is usually impossible due to the fundamental challenge of the state-space explosion problem, even a subset of paths can be used to find bugs or generate a high-coverage test suite [4, 7, 14]. And typically, the more paths are explored, the better the outcome.

With the multitude of paths, performing symbolic execution on a modern machine quickly consumes all available memory. For

---

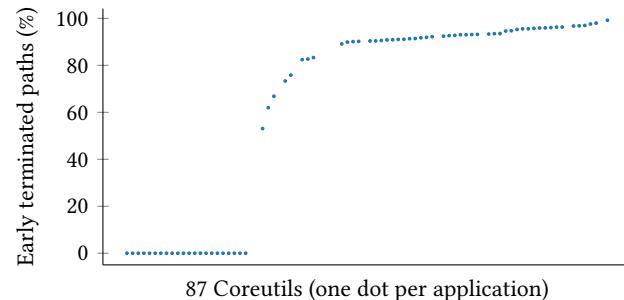
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397360>

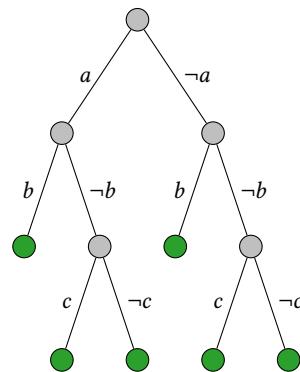


```

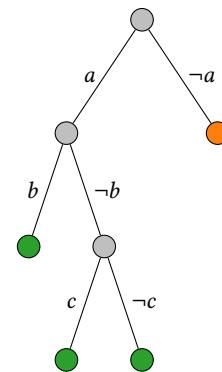
1 int main(void) {
2     int a, b, c; // symbolic
3
4     if (a) { ... }
5     else { ... }
6
7     if (b) { ... }
8     else {
9         if (c) { ... }
10        else { ... }
11    }
12 }

```

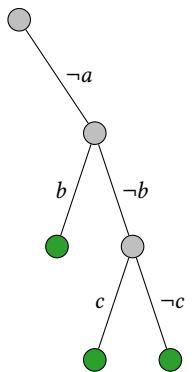
(a) Simple code example



(b) Fully explored tree



(c) Interrupted exploration



(d) Replay with path pruning

Figure 2: A simple code example and associated execution trees.

memoization to larger applications (tens of kLOC) and analysis times (hours and days), we need to overcome several research challenges:

- (1) Real-world applications often interact with the environment. Changes in the environment between the original and replay runs are frequent, and without a robust detection of such changes (*divergences*), re-execution of memoized runs can lead to the exploration of infeasible paths.
- (2) Long runs often involve millions of paths that need to be stored to disk. Storing these paths efficiently while keeping enough information to detect divergences caused by the environment is critical.
- (3) Providing a global view of the stored execution tree is important in many applications, but restoring the whole memoized execution tree consisting of millions of nodes to memory is infeasible.
- (4) Overcoming the restriction of using the same search heuristic during the original and replay runs is important, as it can allow one to be oblivious to the way in which the original tree was created, can speed-up replay, and can allow dynamically changing search heuristics as needed.

In this work, we propose a novel memoization approach for symbolic execution which is designed to overcome the research challenges discussed above. We implement our technique on top of the state-of-the-art symbolic execution engine KLEE [4] and perform an extensive evaluation in which we show that the technique can enable KLEE to run large applications for long periods of time while incurring acceptable space and time overheads.

In the remainder of the paper, we present our approach in Section 2, discuss its implementation in Section 3 and comprehensively evaluate it in Section 4. We then discuss related work in Section 5 and conclude in Section 6.

## 2 APPROACH

Symbolic execution aims to explore all (interesting) execution paths of a program by treating inputs as symbolic. When a symbolic execution engine reaches a conditional statement (e.g. an *if*) whose condition involves symbolic values (*a symbolic branch*), it forks the execution and continues to explore all possible paths. The branch

condition and its negation are attached to the respective sides and form a unique *path condition* along each execution path. The set of all paths form an *execution tree*. An example program and its execution tree are shown in Figures 2a and 2b. Many symbolic execution engines maintain a trie-like data structure to store the execution tree in memory, where intermediate nodes encode symbolic branch feasibility and leaf nodes represent pending execution states. We use the term *execution state*, or simply *state* to denote the representation of a (pending) path in memory.

Constraint solving often incurs a significant computational cost, as it is heavily used to check branch feasibility, to verify safety properties and to generate test cases. Especially when testing applications repeatedly, these costs accumulate quickly as paths have to be re-executed and queries have to be re-solved.

In the following, we present a framework that significantly reduces re-execution times by memoizing solver results (§2.1) and pruning fully-explored subtrees from re-executions (§2.2). Divergence detection (§2.3) ensures that re-executed paths execute the same instructions as their recorded counterparts.

### 2.1 Overview

Memoization is a well-known technique to substitute run-time with storage costs, with results of computationally expensive operations stored and re-used later to avoid re-computations. Prior work [26] memoizes the sequence of choices taken during path exploration.

In our approach, we memoize the execution tree information differently. In general, every node has an *ID*, which is used to associate data with it, and knows the potential IDs of its direct children. We differentiate different node types of the tree, as shown in Figure 3. *Active nodes* are associated with a state, while all *intermediate nodes*—from the root node to an active node—represent symbolic branch decisions that must be made to reach the same state. Therefore, intermediate nodes only represent feasible decisions.

We annotate all *active nodes* with metadata necessary to re-execute a path from its last fork. Specifically, we store solver results, the number of executed instructions, symbolic branches, and basic block hashes to detect divergences (see §2.3), in addition to debug information and basic statistics. We write an active node’s data to a relational database if the associated state is terminated or reaches

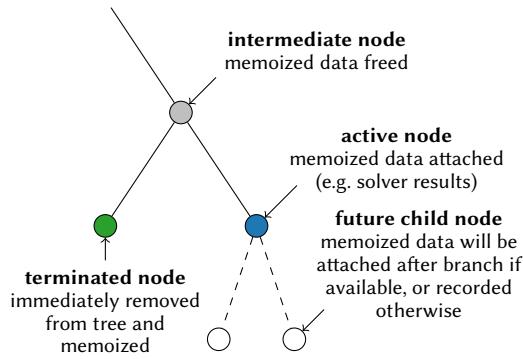


Figure 3: Subtree with different node types.

**Algorithm 1** Satisfiability checking

```

1: global Solver                                ▷ SMT solver
2: function ISSAT(state, condition)
3:   if INREEXECUTIONMODE(state) then
4:     if state.node.solverResults.HASNEXT() then
5:       return state.node.solverResults.GETNEXT()
6:     end if
7:   end if
8:   result ← Solver.ISSAT(state.constraints, condition)
9:   state.node.solverResults.APPEND(result)
10:  return result
11: end function

```

a symbolic branch. In the latter case, the active node becomes an intermediate node with its child nodes being the new active nodes. In either case, the stored data is associated with the node's ID, which on re-execution will provide fast selective access to parts of the execution tree. To keep the memory overhead low, we free the memoized data of intermediate nodes (see Figure 3).

## 2.2 Memoization and Re-execution

During re-execution, we need to associate the memoized data with the new run. A program starts with its initial state and its associated active node with ID 1. As it is a re-execution, data associated with this node can be loaded, particularly solver results up to the next symbolic branch. Such results include those associated with checks for buffer overflows and other errors, as well as queries for concretizing part of the symbolic input (e.g., when calling an external function).

Algorithm 1 shows the function for determining whether a *condition* is satisfiable in a given *state*. If we are re-executing that part of the code and results are memoized, we simply retrieve the next result from the node associated with *state* (lines 3–5). Otherwise, the underlying solver is called (line 8) and the result is recorded in the node (line 9).

The moment a state reaches a symbolic branch, the execution tree needs to be updated. Algorithm 2 shows the code responsible for forking the execution of the current state into up to two child states and the handling of the execution tree nodes. Specifically, function BRANCH takes as input the current *state* and a branch

**Algorithm 2** Branching

```

1: function BRANCH(state, condition)
2:   trueSAT ← ISSAT(state, condition)
3:   falseSAT ← ISSAT(state, ¬condition)
4:   bothFeasible ← trueSAT ∧ falseSAT
5:   if ¬bothFeasible then
6:     ▷ Only one side is feasible, we continue with current state
7:     if trueSAT then
8:       CHECKORAPPENDBRANCH(state, true)          ▷ §2.3
9:       return (state, NULL)
10:    else
11:      CHECKORAPPENDBRANCH(state, false)          ▷ §2.3
12:      return (NULL, state)
13:    end if
14:  else                                     ▷ both branches feasible
15:    CHECKORSETFORK(state)                      ▷ §2.3
16:    trueState ← CREATENEWSTATE(state, condition)
17:    if state.node.trueID then
18:      trueState.node ← INITFROMDB(state.node.trueID)
19:    else
20:      trueState.node ← CREATENEWACTIVENODE()
21:    end if
22:    state.node.trueID ← trueState.node.ID
23:    ...                                         ▷ similarly for falseState
24:    WRITENODETODB(state.node)
25:    FREEDATA(state.node)
26:    return (trueState, falseState)
27:  end if
28: end function

```

*condition* that was encountered during execution and returns a pair of (*trueState*, *falseState*) as result, with either state set to *NULL* if that side of the branch is infeasible.

The algorithm assumes that *state.node* contains a reference to the corresponding execution tree node retrieved from the database. We first determine the satisfiability, in the current *state*, of the *condition* (line 2) and its negation (line 3). If only one side is feasible, we can continue to use the current state and active node and avoid updates of the database (lines 5–13).

If both branches are feasible (line 14), we create a new state (line 16). On line 17, we check whether the corresponding node has its *trueID* set, meaning that the child state where *condition* holds is already in the database. If this is the case, we associate the new state with the corresponding node from the database (line 18). Otherwise, if there is no existing node in the database associated with this state, we create a new node with a unique ID (line 20) and set the *trueID* of the node associated with the current *state* to point to it (line 22).

We repeat the same steps for the other child state (line 23), after which we write the updated *state.node* to the database (line 24), free it from memory (line 25) and return the child states (line 26).

**Switching search heuristics.** Prior work simply memoized the sequence of choices taken during path exploration [26]. Instead, we keep the information about the structure of the execution tree in the database (via the *trueID* and *falseID* relations), which makes our approach completely independent of the chosen exploration

strategy.<sup>2</sup> That is, exploration strategies can vary between the original and re-execution runs. For instance, an interrupted depth-first exploration (Fig. 2c) can be re-executed and completed with a breadth-first traversal (Fig. 2b) without using the constraint solver for the previously explored subtree.

This is useful for various reasons. Firstly, it allows the memoized part of the execution tree to be re-executed much faster. For instance, one might want to use a search heuristic that tries to reach uncovered code during the initial run. However, such heuristics are expensive (e.g. as they may involve shortest paths algorithms), and it would be wasteful to use them during re-execution. Instead, one could use a lightweight heuristic such as depth-first search (DFS) for the memoized parts of the execution, switching to a more effective but expensive heuristic for non-memoized parts.

Secondly, changing the search heuristic can help alleviate memory pressure on re-execution. Suppose that an initial execution is performed using breath-first search (BFS)—which consumes a lot of memory—and then saved to disk. If BFS is used again during re-execution, the same amount of memory would be used. Instead, one could use a heuristic such as DFS which has a small memory footprint during re-execution and only switch to different heuristics for the non-memoized parts.

Thirdly, search heuristics can be used to limit the re-execution to interesting paths that might lead to uncovered code or select narrow subtrees for iterative deepening. While prior work on memoized symbolic execution [26] could also accomplish that, it did so by statically marking nodes for re-execution, which required bringing the entire execution tree into memory.

**Global view and memoization across runs.** Besides being exploration-strategy-agnostic, a big advantage of our memoization framework is the persistent global view of all runs. With every re-execution, newly explored paths are added to the tree stored on disk and provide a more complete picture of the tested application over time. Such a view allows one to reason about an application more thoroughly. Moreover, knowing which paths have been explored, re-executing them to evaluate different properties of an application becomes easier. Metadata of paths that were not fully explored during re-execution is kept, and new paths explored during re-execution automatically start recording new metadata as soon as they progress beyond memoized data. In this way, the execution tree stored on disk can grow across multiple re-execution runs, as we show in our experimental evaluation.

**Path pruning.** An optimisation for re-execution runs is it to remove (prune) all fully-explored subtrees from the exploration. Therefore, we extend the set of metadata to record the termination type for each path (e.g. program exit, bug found, interrupted due to memory pressure) and propagate this information to the parent nodes when a state is terminated. During re-execution, when a path branches, only a single value per branch needs to be checked to determine if its subtree solely contains fully-explored paths. If this is the case, the corresponding branch is terminated (its metadata still stored in the database). That way, only interrupted paths are re-executed, as illustrated in Figure 2d.

---

```

1: symbolic a
2:
3: if a > 10 then      ▷ Spawned state1 follows direction "true"
4: ...
5: end if
6:
7: if a > 5 then      ▷ only direction "true" feasible in state1
8: ...
9: end if
10:
11: if a > 100 then     ▷ both directions feasible in state1
12: ...
13: end if               ▷ state2 spawned, following direction "true"

```

---

Figure 4: Code fragment to illustrate divergence detection.

### 2.3 Divergence Detection

Modern symbolic execution engines mix concrete and symbolic execution [5]. That is, the program under test is allowed to interact directly with the environment, e.g. by calling uninstrumented libraries or performing OS system calls. But during re-execution, changes in the environment—e.g. file timestamps or amount of available memory—can make the path deviate from the original path. Furthermore, because symbolic executors often interleave the execution of different paths, changing the order of execution during re-execution—e.g. by using a different search heuristic—can lead to a different exploration. The reason is that for performance, the execution of single paths is not fully isolated and as such leakage can happen between paths. Finally, symbolic executors often add their own sources of non-determinism, e.g. by using hash tables indexed by concrete memory addresses which may vary across executions.

One key property of symbolic execution is that only feasible paths are explored. To preserve this property, it is necessary that a re-executed path executes the same instruction sequence as its corresponding memoized path. Otherwise, any *divergence* could change the set of collected path conditions and hence invalidate memoized solver results, or even re-use solver results in wrong code locations. In that case, the symbolic executor would explore infeasible paths, potentially leading to false alarms.

To prevent such cases, we memoize and compare against additional safeguarding information: the instruction count, a hash sum of traversed basic blocks, and a vector of symbolic branch directions. Specifically, each execution tree node keeps information summarising the execution between the time the node was first created and the time its associated state branches into two child states. This information consists of:

- (1) A vector of branch directions (*true*, *false*), which starts with the direction that was followed by this state, followed by zero or more entries for any symbolic branches encountered where only one direction was feasible. To make things concrete, consider the code in Figure 4. At line 11, the vector of branches for *state1* is [*true*, *true*] because the state is spawned as a *true* state on line 3 and only the *true* branch is feasible at line 7.

<sup>2</sup>We use the terms *search heuristic* and *exploration strategy* interchangeably.

**Algorithm 3** Divergence detection

---

```

1: function UPDATESAFEGUARDINGDATA(state)
2:   state.node.instructions  $\leftarrow$  state.instructionCounter
3:   state.node.bbHash  $\leftarrow$  state.bbHash
4:   state.node.branches  $\leftarrow$  state.branches
5: end function
6:
7: function CHECKORAPPENDBRANCH(state, branch)
8:   if INREEXECUTIONMODE(state) then
9:     if branch  $\neq$  state.node.branches.getNext() then
10:      MARKSTATEASDIVERGING(state)
11:      UPDATESAFEGUARDINGDATA(state)
12:      state.node.branches.APPEND(branch)
13:    end if
14:   else
15:     state.node.branches.APPEND(branch)
16:   end if
17: end function
18:
19: function CHECKORSETFORK(state)
20:   if INREEXECUTIONMODE(state) then
21:     i  $\leftarrow$  (state.node.instructions = state.instructions)
22:     h  $\leftarrow$  (state.node.bbHash = state.bbHash)
23:     b  $\leftarrow$   $\neg$ state.node.branches.hasNext()
24:     if  $\neg i \vee \neg h \vee \neg b$  then
25:       MARKSTATEASDIVERGING(state)
26:       UPDATESAFEGUARDINGDATA(state)
27:     end if
28:   else
29:     UPDATESAFEGUARDINGDATA(state)
30:   end if
31: end function

```

---

- (2) The number of instructions that were executed before the state forked into two states. In Figure 4, for *state1* this would be the number of instructions executed by that path up to the branch at line 11.
- (3) A cumulative hash of all the basic blocks that were executed before the state forked into two states. In Figure 4, for *state1* this would be a hash of all the basic blocks executed from the start of the program up to and including line 11. A state's basic block hash gets updated whenever a new basic block is reached and relies on a pre-computed map that contains hash sums for all basic blocks.

Function UPDATESAFEGUARDINGDATA in Algorithm 3 shows how we track this data as part of the state and propagate it in lockstep with the associated execution tree node. As part of the memoization, this data is stored when the tree node is written to the database (Alg. 2, line 24).

During re-execution, the run needs to be crosschecked against the information on disk, ensuring that the same branches are taken. The key is to validate this data efficiently. Algorithm 3 shows function CHECKORAPPENDBRANCH, which is called if a symbolic branch has a single outcome (Alg. 2, lines 8 and 11). If the state is in re-execution mode, it checks whether the same memoized branch

---

<pre> 1: <b>external</b> <i>ext</i> 2: <b>symbolic</b> <i>sym</i> 3: 4: <b>if</b> <i>sym + ext</i> <math>&lt;</math> 10 <b>then</b> 5:   OUTPUT("true") 6: <b>else</b> 7:   OUTPUT("false") 8: <b>end if</b> </pre>	<div style="display: flex; justify-content: space-between;"> <span>► concrete</span> <span>► range [0..9]</span> </div>
---	---

---

Figure 5: Code to illustrate divergence detection failure.

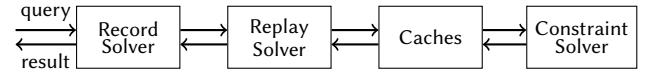


Figure 6: KLEE’s main solving chain elements when memoization is used.

is taken (line 9). If not, it marks the state as diverging (line 10), meaning that all memoized information is discarded and the state continues in recording mode. We then update the safeguarding data (line 11) and append the *branch* to the vector of branches (line 12). In recording mode, the function simply appends the branch to the vector of branches (line 15).

When a state is about to fork into two feasible branches (Alg. 2, line 14), function CHECKORSETFORK from Algorithm 3 checks in re-execution mode whether the same number of instructions have been executed until that point (line 21), the same basic block hash has been computed (line 22) and all recorded branches up to the fork have been consumed (line 23). If not all checks pass (line 24), the state is marked as divergent (line 25) and the safeguarding data is updated (line 26). In recording mode, the function simply updates the safeguarding information associated with the database node (line 29).

Finally, we also perform similar checks when a state terminates, but for space reasons we do not show them in Algorithm 3.

**Limitations.** The algorithm does not detect all divergences. First, it misses rare cases where hash collisions occur, and second, it is not able to detect diverging behaviour that does not change the control flow. An example of the latter is given in Figure 5. When an execution is recorded with an external value *ext* = 0, a re-execution with *ext*  $\geq$  10 re-uses the stored feasibility check results and incorrectly follows the now infeasible true branch. Nevertheless, such situations are quite rare and it is unlikely for our algorithm to completely miss a divergence, given that we perform checks at every single symbolic branch.

### 3 IMPLEMENTATION

We implemented our memoization framework, MoKLEE, on top of KLEE [4] version 1.4,<sup>3</sup> a modern symbolic execution engine for LLVM [16] bitcode. The framework consists of three components: a persistence layer for KLEE’s execution tree (*process tree*) that stores and loads arbitrary metadata transparently, and two new stages in KLEE’s solver chain (Figure 6). The Record Solver appends solver results to nodes whenever paths progress into unmemoized

<sup>3</sup>Some bug fixes and improvements were backported from version 2.0

subtrees and its counterpart, the *Replay Solver*, returns memoized results to the engine. Although both solvers complement each other, they can be used independently.

As shown in Figure 6, KLEE uses caching in front of the actual constraint solver to reduce solving time. It might seem beneficial to place the *Replay Solver* behind the caches to populate them on replay. In practice however, accessing the caches can be expensive—for instance, a 7-day run of *split* from *Coreutils* spends more than 99% in the caches. MoKLEE still supports both solver placements but it is an essential design decision to override the caches for faster re-execution times. Progressing states still benefit from the fact that caches typically fill quickly.

Another important implementation aspect concerns solver queries that contain memory addresses. Our framework does not implement means for deterministic memory allocation and forwards affected queries to the solving chain without memoizing them.

A large part of the engineering effort also went into other parts of KLEE, to make it more reliable on long-running experiments. For instance, initially our implementation was not capable of re-executing even 1% of the instructions of *readelf*. The reason for that was its heavy use of function pointers and KLEE’s non-deterministic assignment of these addresses. KLEE internally re-uses addresses of the corresponding LLVM function objects which are allocated differently after restarting KLEE. We modified KLEE in such a way that it assigns the same addresses in every run.

Some mitigation strategies are already implemented in KLEE, such as providing a fixed set of environment variables to the program under test and, most notably, deterministic memory. This mechanism does not ensure that pointers have the same value in re-executions, but at least have the same ordering. Internally, KLEE pre-allocates a memory map and incrementally assigns memory from that map for program allocations. The disadvantage of the deterministic mode is that allocated space is never freed, rendering it impractical for many applications. Hence, we refrained from using this mode although it reduces the occurrences of divergences significantly.

## 4 EVALUATION

We have extensively evaluated our approach on 93 Linux applications. The evaluation section is structured as follows. Section 4.1 discusses the experimental setup, Section 4.2 presents the runtime savings achieved during re-execution, Section 4.3 reports the space and runtime overhead of our framework, and Section 4.4 discusses divergences and their impact in re-executions. Finally, Section 4.5 shows how our framework enables effective iterative deepening with symbolic execution, and Section 4.6 how it allows applications to run for days in a row.

An artifact with our experiments is available at <https://srg.doc.ic.ac.uk/projects/moklee/>.

### 4.1 Experimental Setup

For our experiments, we use a set of homogeneous machines with Intel Core i7-4790 CPUs at 3.6 GHz, 16 GiB of RAM, and 1 TB hard drive (7200 rpm). All experiments run in Docker containers that use the same operating system as the host machines (Ubuntu 18.04). If not stated otherwise, we use KLEE’s default memory limit of 2 GB.

MoKLEE is built against LLVM 3.8 and uses Z3 [8] as SMT solver with a timeout of 10 s.

As benchmarks, we selected a variety of different applications: from the GNU software collection we selected the *Coreutils* suite, *diff*, *find* and *grep*. These are non-trivial systems applications used by millions of users. With *libspng*, *readelf* and *tcpdump*, we additionally selected applications that process more complex input formats such as images, binaries and network packets.

In more detail, the applications are:<sup>4</sup>

**GNU Coreutils** [10] (version 8.31; 66.2k LOC) provide a variety of tools for file, shell and text manipulation. Used in the paper introducing KLEE [4], they have become the de-facto benchmark suite for KLEE-based tools. The suite consists of 106 different applications, from which we excluded five tools that are very similar to *base64* (*base32*, *sha1sum*, *sha224sum*, *sha384sum*, *sha512sum*) and one tool (*cat*) whose execution with KLEE runs out of memory after a few seconds due to large internal buffers in recent versions. Furthermore, we excluded 13 tools (*chcon*, *chgrp*, *chmod*, *chown*, *chroot*, *dd*, *ginstall*, *kill*, *rm*, *rmdir*, *stdbuf*, *truncate*, *unlink*) that behave non-deterministically under symbolic execution, as executing paths through these tools can affect the execution of other paths or even Docker’s and KLEE’s behaviour without additional isolation in place (which is possible, but whose implementation is orthogonal to the goals of the project). This leaves us with a total of 87 tools. As in the original KLEE paper [4], we patched *sort* to reduce the size of a large buffer. *Coreutils* link against the GNU Portability Library (as do *diff*, *find* and *grep* discussed below). We use the same revision of this library (git #d6af241; 484.5k LOC) in all applications.

**GNU diff** from GNU Diffutils [11] (version 3.7; 7.9k LOC) is a command-line tool to show differences between two files.

**GNU find** from GNU Findutils [12] (version 4.7.0; 15.7k LOC) searches for files in a directory hierarchy.

**GNU grep** [13] (version 3.3; 4.1k LOC) searches for text in files that matches specified regexes.

**GNU readelf** from GNU Binutils [9] (version 2.33; 78.3k LOC and 964.4k of library code<sup>5</sup>) displays information about ELF object files.

**libspng** [18] (git #2079ef6; 4k LOC) is a library for reading and writing images in the Portable Network Graphics (PNG) file format. We wrote a small driver (24 LOC) that reads a symbolic image and links with the zlib [23] compression library (version 1.2.11; 21.5k LOC).

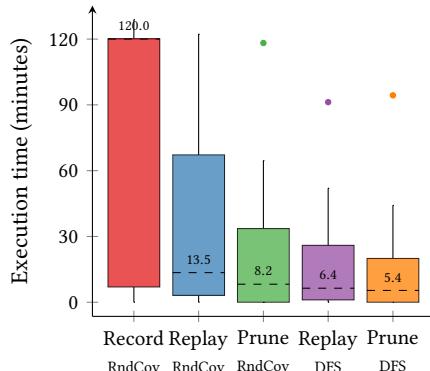
**tcpdump** [24] (version 4.9.3; 77.5k LOC) analyses network packets. We link against git #f030261 of its accompanying *libpcap* library (44.6k LOC).

### 4.2 Speed of Re-execution

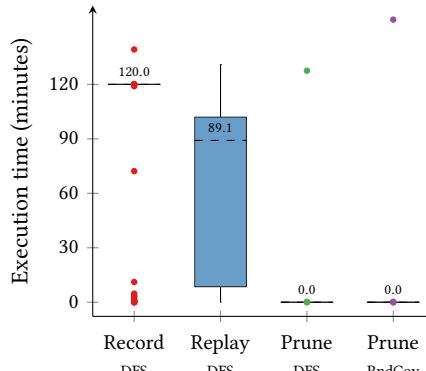
MoKLEE implements two mechanisms to reduce runtime costs of re-executions: 1) memoization of constraint solver results, and 2) path pruning. To evaluate their effectiveness, we run MoKLEE in recording mode on our 93 benchmarks for 2 h and measure the time it

<sup>4</sup>Lines of code (LOC) are reported by *scc* [1] and are for the whole application suites from which our benchmarks come, as deciding which lines of code “belong” to an individual application is difficult, if at all possible.

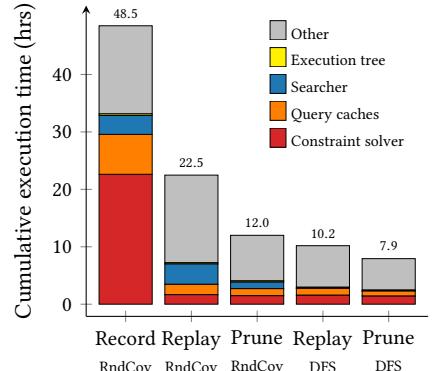
<sup>5</sup>We only consider lines in the *binutils* folder and the following dependencies: *libbfd*, *libctf*, *libltdl*, and *libpcodes*.



**Figure 7: Distribution of execution times for 37 Coreutils, when *RndCov* is used during the initial run.**



**Figure 8: Distribution of execution times for 74 Coreutils, when *DFS* is used during the initial run.**



**Figure 9: Cumulative execution times for different subsystems of MoKLEE for the runs in Figure 7.**

takes to re-execute the same execution trees. As the selected exploration strategy has an impact on the effectiveness of each mechanism, we evaluate our implementation with two different heuristics: the memory-friendly depth-first search (*DFS*) and KLEE’s default strategy, a combination of random path selection and coverage-guided search (*RndCov*).

We set the memory limit to 2 GB; if this limit is reached, states are terminated prematurely to stay within it. Although the memory overhead of our framework is small, it might be enough to trigger an earlier state termination during a full replay with a different heuristic. When this happens, fewer paths are replayed, making it meaningless to compare the speed of re-execution. To mitigate the problem, we slightly increase the memory limit of re-execution runs from 2 GB to 2.2 GB and terminate states as soon as they are fully replayed. (Note that we only do this for the purpose of this experiment, none of those adjustments are needed when using MoKLEE in a real setting, where one would typically only restore states of interest instead of performing a full replay.)

**Coreutils with *RndCov* during recording.** We first memoize 87 Coreutils with the default exploration strategy (*RndCov*) and re-execute these runs with path pruning disabled (*replay*) or enabled (*prune*) using each of the two search strategies (*DFS*, *RndCov*). To keep different re-executions comparable, we remove all tools that diverge or terminate states early due to memory pressure in at least one of the runs.

The results for the 37 remaining tools are shown in Figure 7. The median execution time decreases significantly from 120 min, our chosen timeout, to 13.5 min when re-executed with the same exploration strategy and even further to 8.2 min when path pruning is enabled. 11 applications that terminate within our time limit benefit most from path pruning: the whole tree is pruned and not a single instruction needs to be re-executed. When the less expensive *DFS* exploration strategy is used during re-execution, the median execution times are further reduced to 6.4 min and 5.4 min respectively. The outlier in this graph is *yes*, whose core functionality is an infinite loop with no constraint solving, and thus does not benefit from memoization (including path pruning,

due to the broad exploration of the *RndCov* heuristic used during recording).

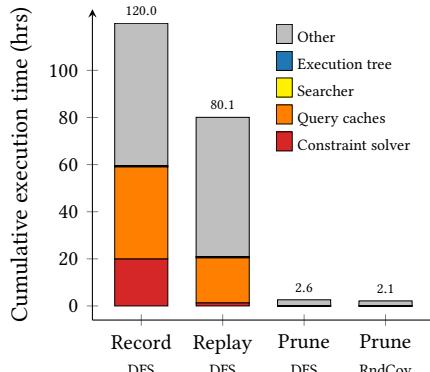
**Coreutils with *DFS* during recording.** We repeat this experiment using *DFS* in the recording run. As before, to keep different re-executions comparable, we remove all tools that diverge or terminate states early due to memory pressure in at least one of the runs. The results for the 74 remaining tools are shown in Figure 8. Note that we do not report results for the *RndCov* re-execution without path pruning: while *DFS* shapes an execution tree that consists of large fully-explored subtrees along an active path, a *RndCov* exploration without path pruning of that same tree creates too many states, and KLEE terminates most of them due to memory pressure.

Without path pruning, recording and replaying the execution tree with *DFS* results in lower savings than when *RndCov* is used in both stages: this is because the *DFS* run spends less time solving constraint queries. However, when path pruning is used, the re-execution time is usually under a minute, as the shape of the *DFS*-generated execution tree is ideal for pruning: re-executions only need to follow the active path and can prune all adjacent subtrees, reducing the re-execution time significantly.

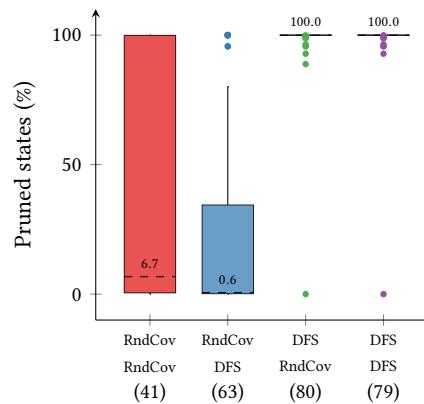
The outlier taking more than 2 h during recording is *split*; we are currently investigating the reasons. The outlier that exceeds the timeout of 2 h during re-execution is *shred*. This application ends up in a tight loop in *DFS* mode, creating more than  $1.9 \times 10^{10}$  instructions with a negligible amount of constraint solving (0.1%).

**Impact on MoKLEE components.** To show which components of MoKLEE benefit most from the memoization during re-execution, we break down the cumulative execution time of each experiment configuration into the time spent in each component.

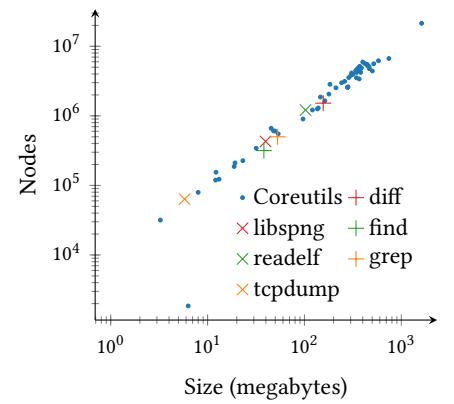
The results are shown for *RndCov* in Figure 9 and for *DFS* in Figure 10. As expected, memoization significantly reduces the time spent in constraint solving and also the time spent in the query caches, as our framework is placed in front of the solving chain. Comparing Figure 9 and 10 one can see that *RndCov* spends significant time in the searcher, due to its expensive coverage calculations and tree traversals. By contrast, *DFS* spends insignificant time in the searcher.



**Figure 10:** Cumulative execution times for different subsystems of MoKLEE for the runs in Figure 8.



**Figure 11:** Distribution of pruned states across all path-pruning runs (number of *Coreutils* in each experiment in parentheses).



**Figure 12:** Execution tree sizes on disk for 66 runs that reach the 2 h limit. Sizes range between 3.10 MiB (*pathchk*) and 1.50 GiB (*echo*).

Finally, we report the distribution of pruned states for all re-executions in Figure 11. The more states can be pruned, the fewer instructions have to be re-executed. Obviously, the set of pruned states in a *complete* re-execution is independent of the search strategy used during re-execution. The difference between *RndCov/RndCov* and *RndCov/DFS* is merely caused by the different sets of non-diverging applications. The outlier in the experiments with a recorded *DFS* execution is *yes*. By executing an endless loop, *yes* creates a very deep path that never terminates and hence can't be pruned.

**Non-Coreutils applications.** For these applications, we memoize 2 h runs only with *RndCov* and use both exploration strategies on re-execution. Table 1 shows for each run the number of recorded instructions, the time needed by the re-execution, the number of instructions successfully replayed and the number of diverging states.

*libspng* and *readelf* have no divergences. Here, our implementation significantly reduces the execution time to between 1.39% and 15% of the initial execution time.

*diff*, *grep* and *tcpdump* suffer from divergences, but MoKLEE is able to re-execute most of the instructions in the recorded runs. In all three cases, more than 99% of the instructions could be replayed, in a time ranging from 3.45% to 25.76% of the initial execution time.

*find* has several thousand divergences, and thus MoKLEE is only capable of replaying 78.07% (79.27%) of the instructions in 9.87% (24.29%) of the initial execution time for *DFS* (*RndCov*). We will discuss this case and possible mitigations in Section 4.4.

### 4.3 Space and Runtime Overhead

Memoization does not come for free: execution tree nodes use additional memory during runtime, large execution trees require significant disk space, and there is additional computational overhead for managing the execution tree and the database. A big advantage of our implementation is the trie-like structure: states share common prefixes that are only stored once in memory and on disk.

Fortunately, the additional memory overhead of memoization is negligible. As discussed before, paths that are skipped by our

**Table 1:** Results for re-executions of memoized runs (2 h, *RndCov*) with different exploration strategies and path pruning disabled. Although some tools have diverging states, most instructions could be re-executed successfully.

Tool	Search	Recorded Instrs (M)	Re-execution Time (%)	Instrs (%)	Diverging states
diff	<i>DFS</i>	65.1	22.01	99.13	64
diff	<i>RndCov</i>	65.1	25.76	99.27	60
find	<i>DFS</i>	1,105.3	9.87	78.07	9,518
find	<i>RndCov</i>	1,105.3	24.29	79.27	8,633
grep	<i>DFS</i>	32.5	4.45	99.99	3
grep	<i>RndCov</i>	32.5	5.86	99.99	3
libspng	<i>DFS</i>	22.0	1.39	100.00	0
libspng	<i>RndCov</i>	22.0	2.44	100.00	0
readelf	<i>DFS</i>	99.4	9.01	100.00	0
readelf	<i>RndCov</i>	99.4	15.00	100.00	0
tcpdump	<i>DFS</i>	10.0	3.45	99.96	2
tcpdump	<i>RndCov</i>	10.0	3.68	99.95	3

path-pruning mechanism are never attached to the execution tree and terminated subtrees are freed from memory immediately. Every intermediate node (Fig. 3) consumes 24 B extra per node and active and leaf nodes require an additional 208 B without any progress. This includes some statistics and debugging information. The exact amount gathered throughout execution in leaf nodes depends on the number of queries issued and the number of symbolic branches taken by the respective state. Each memoized satisfiability query result requires two bits. For divergence detection, we store one bit for the single outcome of a symbolic branch point.

The required disk space is roughly linear to the number of stored nodes, as tree nodes only keep enough information for a state to reach the next node. Figure 12 shows the storage sizes for all

*RndCov* memoization runs from Section 4.2 which reached the 2 h timeout. Sizes vary between 3.10 MiB (*pathchk*) and 1.50 GiB (*echo*). On average, a single node requires 85 B of disk space.<sup>6</sup> The outlier in the bottom-left corner of Figure 12 is again *yes*. The core of *yes* is basically a tight print loop guarded by a symbolic condition with one feasible branch. As a result, it branches fewer than 1000 states in 2 h but some of the corresponding nodes have to store more than 100 million branch decisions for divergence detection. However, this type of behaviour is not representative of other applications.

To measure the runtime overhead, we use the *DFS* re-executions without path pruning (*replay*) from Figure 10, where *DFS* was also used during recording. This is close to the worst case for memoization: the paths reached with *DFS* are typically deeper such that more intermediate nodes have to be memoized and stored to disk more often. Still, the computational overhead for a path explored via *DFS* is typically smaller as the cache utilisation for constraint solving is much higher reducing the overall solving costs while at the same time the search heuristic has almost no overhead. This leaves only little room to offset performance using memoization.

In the first experiment, we measure the cumulative time spent in managing the metadata for memoization during runtime and time spent in reading and writing the execution tree to disk (I/O) for both the recording and replaying runs. Figure 13 shows the distribution of these times relative to the overall execution times. As can be seen, this time is never higher than 2% during recording and replay, with the median values of only 0.19% and 0.36% respectively. Moreover, our experiment setup uses traditional rotating hard disks. The overhead could be reduced further by using faster solid-state drives (SSDs).

To quantify the impact of I/O operations, we perform a run identical to the recording run, except that we disable storing the tree to disk. We keep the 42 applications that behave in the same way when memoization is disabled. Memoization without I/O adds a median overhead of only 1.22%, which is similar to what we observed when measuring the extra time spent writing the execution tree to disk in recording mode. The overhead in this experiment varies between -1.9% and 5.5%. The negative overhead is likely due to some measurement noise arising from running our experiments on a cluster of machines and any unexpected influences of our implementation on machine-level caches.

#### 4.4 Divergences

In this section, we assess how common divergences are and how they affect re-execution. We first note that some applications suffer from divergences regardless of the exploration strategy used. Many of these applications rely on the execution environment. For instance, they print the system time (*date*), show free (*df*) or used (*du*) disk space or the time for which the system has been running (*uptime*).<sup>7</sup>

We again re-use the experiments from Section 4.2, with path pruning disabled and consider all four combinations of exploration

<sup>6</sup>For the sake of simplicity we measure the database file size including metadata.

<sup>7</sup>Although our experiments run in Docker containers on a set of homogeneous machines using the same sandbox directories, we did not try to fully virtualize the execution to provide identical execution environments. Such virtualization could reduce the number of divergences but it would necessitate significant engineering effort and would not take advantage of the mixed concrete-symbolic execution paradigm at the core of dynamic symbolic execution [5].

strategies between recording and re-execution runs. We remove applications that on replay terminate states due to memory pressure, as our focus here is on runs where the replay would incorrectly follow infeasible paths due to divergences, rather than the case where different paths are executed due to memory pressure.

**Coreutils.** Figure 14 shows the data for *Coreutils*. The figure plots the distribution of the number of *lost instructions*, which are instructions in subtrees that were discarded when a divergence occurred. Execution trees resulting from *RndCov* explorations are typically less deep than their *DFS* counterparts. This means that in case of divergences only small incomplete subtrees are removed for *RndCov* instead of large fully-explored subtrees for *DFS*. Figure 14 clearly shows this difference: whereas a *RndCov/RndCov* combination loses only a small number of instructions (median 6.22%), a *DFS/DFS* run not only has a higher median value (27.61%) but also a much larger maximum (99.95% vs. 61.38%).

**libspng** and **readelf** do not encounter divergences.

**diff**, **grep**, and **tcpdump** have a small number of diverging states, as shown in Table 1. Despite these divergences, more than 99% of instructions could be re-executed successfully.

**find** suffers most from divergences and only re-executes 78.07% of instructions with *DFS* and 79.27% with *RndCov*. Most of the divergences occur in its `memmove` function, which we suspect are due to different memory allocation schemes between the record and replay runs. To confirm this hypothesis, we re-ran *find* with KLEE’s deterministic allocation mode (`-allocate-determ`, see §3 for a discussion of why this mode is often impractical), with an additional 1 GB of memory. As expected, the number of re-executed instructions increased significantly, to 96.96% for *DFS* and 99.17% for *RndCov*.

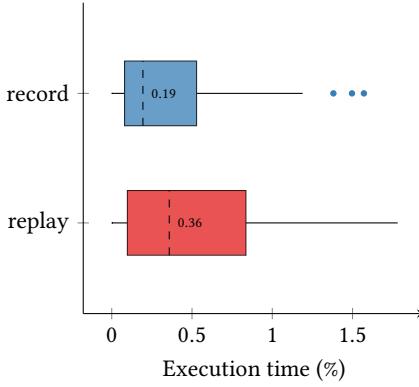
Finally, we re-emphasise the fact that divergences have a much worse impact than losing memoized instructions. Without detecting them, symbolic execution can start exploring infeasible paths, potentially wasting a lot of time without making any meaningful progress and even generating false positives in the process.

#### 4.5 Iterative Deepening

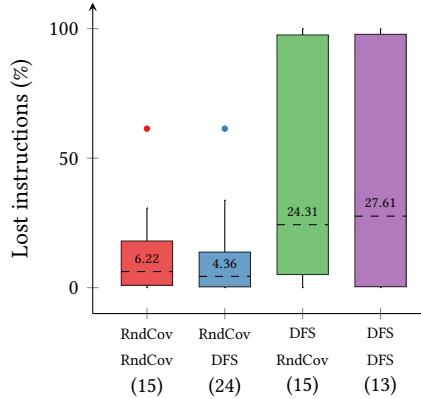
A natural use case for memoization is iterative deepening, which is able to mimic *BFS* exploration with much lower memory consumption by *repeatedly exploring* a program with *DFS* up to a certain (increasing) depth in the execution tree.

For this experiment, we select and run the set of non-*Coreutils* applications with *BFS* exploration until KLEE’s default memory limit of 2 GB is exceeded. At this point, we cannot run in *BFS* mode anymore without losing paths. Instead, we employ iterative deepening, making use of the memory-friendly *DFS* in each iteration. Two of the benchmarks, *diff* and *readelf*, had very long runtimes and we decided to reduce the solver timeout in both cases to 1 s. Also, we excluded *find* from the selected benchmarks because of its large number of divergences (see Table 1).

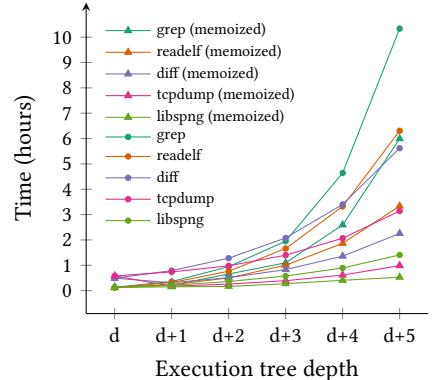
As starting depth values for iterative deepening, we choose the minimum tree depths for which states are terminated due to memory pressure ( $d = 22$  for *diff*, 13 for *grep*, 42 for *libspng*, 22 for *readelf* and 19 for *tcpdump*). We iteratively increase the depths by five more levels (to  $d+5$ ), once without and once with memoization (including path pruning).



**Figure 13:** Execution time spent in tree operations for the DFS recording run and its replay from Figure 10, which is close to the worst-case scenario in terms of runtime overhead.



**Figure 14:** Distribution of lost instructions due to divergences on re-execution (no path pruning, number of Coreutils in each experiment in parentheses).



**Figure 15:** Iterative deepening with and without memoization. For each application and depth, we show the time it takes to perform the run with and without memoization.

Figure 15 shows how the memoized runs compare with the non-memoized ones. For all five applications, memoization significantly reduces the execution time, typically by more than 50%, which represents several hours of execution for these experiments. No divergences were observed in the re-execution runs.

**Validity.** It is not guaranteed in this setup that the memoized and non-memoized experiments execute the same instructions, especially with a small solver timeout and different usage of solver caches. With path pruning during memoization and no execution tree recorded in the unmemoized experiment, there is no practical way to directly compare both executions. Therefore, we use another experiment as a proxy for comparison. We repeat the run with the highest depth ( $d + 5$ ) and memoize the execution tree (*record-only*). We then compare 1) the number of instructions of the non-memoized and record-only run, and 2) the execution trees of the record-only and memoized runs.

The number of instructions executed in record-only runs vary between 99.97% (*grep*) and 100.2% (*readelf*) in comparison to their non-memoized counterparts. Assuming that this reflects a high similarity between both runs, we finally compare the nodes of the execution trees of the record-only and memoized runs. The trees for *tcpdump* are identical, while the other trees have a high number of identical nodes (*readelf* 98.0%, *libspng* 99.3%, *grep* 99.94%, and *diff* 100.0%). Therefore, we believe the comparison in Figure 15 is meaningful.

## 4.6 Long-Running Experiments

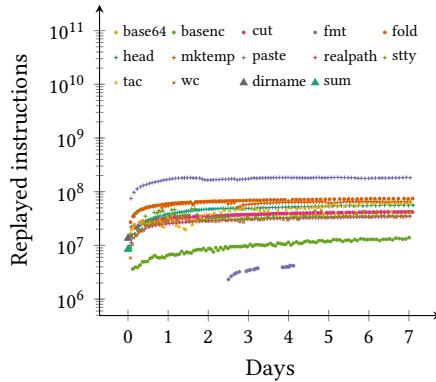
Our final set of experiments demonstrates that our framework enables symbolic executors to run for very long periods of time while continuing to explore new paths.

We focus on the applications discussed in the introduction, for which KLEE configured with a 2 GB memory limit completely runs out of paths before the 2 h limit is reached, because too many states are killed prematurely due to memory pressure. For our *RndCov*

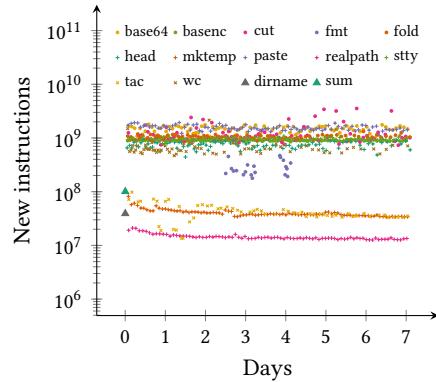
recording run above, these applications are *base64*, *basenc*, *cut*, *dirname*, *fmt*, *fold*, *head*, *mkttemp*, *paste*, *realpath*, *stty*, *sum*, *tac* and *wc*. As discussed before, no matter how much time one has at their disposal, KLEE won't be able to explore more than a certain number of paths in such cases. While increasing the memory limit could help, this is often an ineffective workaround, as important paths may still be lost on memory pressure, and symbolic execution may run out of paths at a later time. Furthermore, a low memory limit is often advantageous in a cloud deployment, where renting machines with little memory is more cost-effective. Below, we show that using our approach, one can run these applications for days, using KLEE's default memory limit of 2 GB, without losing any paths and continuously exploring new ones.

Starting with the memoized runs from the first experiment (where symbolic execution ran out of paths), we re-execute (with path pruning enabled) each of the 14 applications repeatedly from their previous recorded execution until the cumulative execution time exceeds seven days. Every time we restart the application, we run MoKLEE without a timeout until it runs out of paths again and terminates. We use the default *RndCov* heuristic in each run.

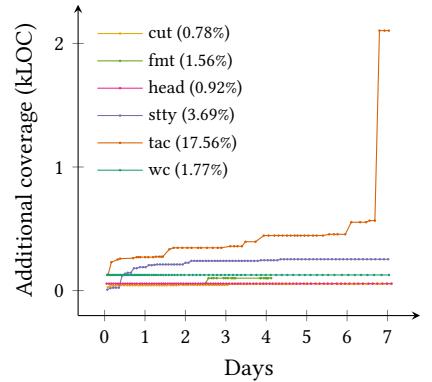
Under this setup, we had to restart the applications between 20 times (for *fmt*) and 105 times (for *basenc*), with two exceptions: *dirname* and *sum* finish normally (without early path termination) after their first re-execution. Figures 16 and 17 show the number of replayed instructions and the number of new (unmemoized) instructions over time. Each point corresponds to one run. The figures show that our memoization extension is effective—each restarted run is able to progress the exploration and execute up to two orders of magnitude more new (unmemoized) instructions than already memoized ones. For six of these applications, this translates into new coverage. Figure 18 plots the additional coverage over time for these applications. After seven days, 4 of these applications achieved around 1–2% extra coverage, *stty* 3.69% more, and *tac* an impressive 17.56% more. Even more interestingly in the case of *tac*, most extra coverage is achieved in the last day, showing that in



**Figure 16: Replayed instructions per re-execution run.**<sup>8</sup>



**Figure 17: New (unmemoized) instructions per re-execution run.**<sup>9</sup>



**Figure 18: Additionally covered bitcode lines. Relative increase over recording runs given in parentheses.**

some configurations, long-running runs are needed to achieve high coverage.

After seven days, the execution tree sizes vary between 0.6 GiB (*tac*) and 37.7 GiB (*wc*), which are reasonable values. These encode from 6.7M to over 495M nodes. Only *mkttemp* encountered divergences (up to 9 per run) in this experiment.

## 5 RELATED WORK

Some forms of memoization, persistent execution trees, and path pruning during re-execution have been proposed in prior work. As discussed in the introduction, our work extends the excellent idea of memoized symbolic execution [26] to make it possible to detect divergences, store long-running paths without having to bring the entire execution tree back into memory and overcoming the restriction of using the same search heuristic in the record and replay runs. All these contributions make it possible for MoKLEE to scale from the type of short runs on which memoized symbolic execution was previously evaluated (on the order of minutes, with small memoized trees) [22, 26] to very long runs (of hours and even days, with trees of millions of nodes).

*Mayhem* [7] introduced a combination of offline and online symbolic execution. When Mayhem runs out of memory, it terminates selected states after creating complete snapshots containing the path predicate and various statistics. Later, when more resources are available, states are brought back into memory from their snapshots and only associated concrete paths are re-executed. The authors give an average size of 30 kB per snapshot for *echo*. Considering that in our experiments *echo* created a large file on disk (1.50 GiB) with an average of 151 B per state and 10.5 million early terminated states, Mayhem-style checkpointing does not scale in our context. A *virtualization layer* in Mayhem intercepts and emulates system calls, preventing interferences between states and most likely divergences in re-executions.

<sup>8</sup>*fmt* has one extra point after 9.9 days with value  $4.2 \times 10^6$ , which we do not show for better readability of the graph.

<sup>9</sup>*fmt* has one extra point after 9.9 days with value  $1.6 \times 10^8$ , which we do not show for better readability of the graph.

*Cloud9* [3] is a KLEE-based symbolic executor. One of its features is the distributed exploration of programs under test. For that, recorded execution path prefixes for candidate nodes are sent to workers which then re-execute those paths and resume exploration from such nodes. During re-execution of received path prefixes, all queries have to be re-solved by a constraint solver. Divergences are not detected. Our implementation currently has only limited support for distributed exploration. We always transfer complete trees between workers and run experiments incrementally.

*FuzzBall* [20], a symbolic execution engine for x86 binaries, explores a single path at a time. When a path terminates, a new one gets explored from the program start. An in-memory execution tree is mainly used to keep track of explored subtrees and to prevent expensive feasibility checks when reaching a symbolic branch. In case the tree becomes too large for an in-memory representation, users can optionally enable a (much slower) on-disk mode.

Orthogonal to memoization are *persistent constraint solution caches* [15, 25]. In our experiments we made three observations: 1) constraint solving time is often dominated by KLEE’s caches rather than the underlying constraint solver (see Figure 10), 2) caches fill up quickly, and 3) most queries are solved quickly. It was an essential design decision to place our re-execution mechanism in front of the caches to reduce re-execution times significantly.

*Seeding* is another orthogonal approach to guide symbolic execution and reduce solving time [14, 19]. A seed is a concrete input (test case) for a program under test that describes a single execution path in the execution tree. Along such a path, the execution engine can omit branch-feasibility checks as a feasible input is already known. The main weakness of using seeding for memoization is that a large number of seeds would need to be stored, making the approach impractical for large execution trees. Instead of generating test cases for every execution state, SynergiSE [21] exploits the implicit ordering of tests and uses bordering tests to describe unexplored or feasible subtrees. While this representation can significantly reduce space requirements compared to keeping one seed per leaf node, it relies on a fixed search order and doesn’t provide the global view of a full execution tree.

Outside of symbolic execution, memoization has been proposed in other domains. For instance, various forms of incremental and cooperative model checking rely on reusing analysis results previously saved to disk [2, 6, 17].

## 6 CONCLUSION

In this paper, we have presented an approach which enables symbolic execution to run indefinitely on large applications while continuing to explore new paths. Our approach is based on memoization, enhanced with support for divergence detection, switching search heuristics between record and replay time, and efficient storage which preserves the structure of the execution tree. We implemented our approach in MoKLEE, an extension of the popular symbolic execution engine KLEE, and performed an extensive evaluation on 93 Linux applications. Our evaluation shows practical space and runtime overheads, high re-execution speed, effective divergence detection and applicability to iterative deepening and long-running symbolic execution analysis.

## ACKNOWLEDGEMENTS

We thank Khoo Wei Ming and Cho Chia Yuan for their feedback on this project. We also thank Timotej Kapus, Jordy Ruiz and the anonymous reviewers for their comments on the paper.

This research has received funding from the DSO National Laboratories, Singapore, and the EPSRC, UK through grants EP/R011605/1 and EP/N007166/1. This project has also received funding from European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 819141).

## REFERENCES

- [1] Ben Boyter. 2019. *Sloc Cloc and Code (scc)*. <https://github.com/boyter/scc>
- [2] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A technique to pass information between verifiers. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE’12)*.
- [3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proc. of the 6th European Conference on Computer Systems (EuroSys’11)*.
- [4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08)*.
- [5] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90.
- [6] Rodrigo Castaño, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2017. Model Checker Execution Reports. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE’17)*.
- [7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P’12)*.
- [8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08)*.
- [9] GNU. 2019. *GNU Binutils*. <https://www.gnu.org/software/binutils/>
- [10] GNU. 2019. *GNU Coreutils*. <https://www.gnu.org/software/coreutils/>
- [11] GNU. 2019. *GNU Diffutils*. <https://www.gnu.org/software/diffutils/>
- [12] GNU. 2019. *GNU Findutils*. <https://www.gnu.org/software/findutils/>
- [13] GNU. 2019. *GNU Grep*. <https://www.gnu.org/software/grep/>
- [14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS’08)*.
- [15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA’15)*.
- [16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO’04)*.
- [17] Steven Lauterburg, Ahmed Sobieh, Darko Marinov, and Mahesh Viswanathan. 2008. Incremental State-space Exploration for Programs with Dynamically Allocated Data. In *Proc. of the 30th International Conference on Software Engineering (ICSE’08)*.
- [18] libspng [n.d.]. *libspng*. <https://github.com/randy408/libspng>
- [19] Paul Dan Marinescu and Cristian Cadar. 2012. make test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE’12)*.
- [20] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’12)*.
- [21] Rui Qiu, Sarfraz Khurshid, Corina S. Păsăreanu, Junye Wen, and Guowei Yang. 2018. Using Test Ranges to Improve Symbolic Execution. In *Proc. of the 10th International Conference on NASA Formal Methods*.
- [22] Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2015. Compositional Symbolic Execution with Memoized Replay. In *Proc. of the 37th International Conference on Software Engineering (ICSE’15)*.
- [23] Greg Roelofs and Mark Adler. [n.d.]. *zlib*. <https://zlib.net/>
- [24] The tcpdump team. [n.d.]. *tcpdump*. <https://www.tcpdump.org/>
- [25] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE’12)*.
- [26] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA’12)*.



# Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach

Yiling Lou\*

HCST, CS, Peking University  
Beijing, China  
yiling.lou@pku.edu.cn

Haotian Zhang

Ant Financial Services Group  
Hangzhou, China  
jingyun.zht@antfin.com

Ali Ghanbari

Xia Li

Lingming Zhang<sup>†</sup>  
University of Texas at Dallas  
Texas, USA

{Ali.Ghanbari,Xia.Li3,lingming.zhang}@utdallas.edu

Dan Hao<sup>†</sup>

Lu Zhang

HCST, CS, Peking University  
Beijing, China  
{haodan,zhanglucs}@pku.edu.cn

## ABSTRACT

A large body of research efforts have been dedicated to automated software debugging, including both automated fault localization and program repair. However, existing fault localization techniques have limited effectiveness on real-world software systems while even the most advanced program repair techniques can only fix a small ratio of real-world bugs. Although fault localization and program repair are inherently connected, their only existing connection in the literature is that program repair techniques usually use off-the-shelf fault localization techniques (e.g., Ochiai) to determine the potential candidate statements/elements for patching. In this work, we propose the *unified debugging* approach to unify the two areas in the other direction for the first time, i.e., *can program repair in turn help with fault localization?* In this way, we not only open a new dimension for more powerful fault localization, but also extend the application scope of program repair to all possible bugs (not only the bugs that can be directly automatically fixed). We have designed ProFL to leverage patch-execution results (from program repair) as the feedback information for fault localization. The experimental results on the widely used Defects4J benchmark show that the basic ProFL can already at least localize 37.61% more bugs within Top-1 than state-of-the-art spectrum and mutation based fault localization. Furthermore, ProFL can boost state-of-the-art fault localization via both unsupervised and supervised learning.

\*This work was mainly done when Yiling Lou was a visiting student in UT Dallas.  
HCST is short for Key Lab of High Confidence Software Technologies, MoE, China.  
<sup>†</sup>Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA  
© 2020 Association for Computing Machinery.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397351>

Meanwhile, we have demonstrated ProFL's effectiveness under different settings and through a case study within Alipay, a popular online payment system with over 1 billion global users.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Automated Program Repair, Fault Localization, Unified Debugging

### ACM Reference Format:

Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397351>

## 1 INTRODUCTION

Software bugs (also called software faults, errors, defects, flaws, or failures [74]) are prevalent in modern software systems, and have been widely recognized as notoriously costly and disastrous. For example, in 2017, Tricentis.com investigated software failures impacting 3.7 Billion users and \$1.7 Trillion assets, and reported that this is just scratching the surface – *there can be far more software bugs in the world than we will likely ever know about* [70]. In practice, software debugging is widely adopted for removing software bugs. However, manual debugging can be extremely tedious, challenging, and time-consuming due to the increasing complexity of modern software systems [68]. Therefore, a large body of research efforts have been dedicated to automated debugging [8, 34, 52, 59, 68].

There are two key questions in software debugging: (1) *how to automatically localize software bugs to facilitate manual repair?* (2) *how to automatically repair software bugs without human intervention?* To address them, researchers have proposed two categories of techniques, *fault localization* [5, 14, 30, 42, 51, 80, 81] and *program repair* [32, 36, 38, 44, 45, 60, 61, 72]. For example, pioneering

spectrum-based fault localization (SBFL) techniques [5, 14, 30] compute the code elements covered by more failed tests or less passed tests as more suspicious, and pioneering mutation-based fault localization (MBFL) techniques [51, 55, 81] inject code changes (e.g., changing `>` into `=`) based on mutation testing [17, 26] to each code element to check its impact on test outcomes; meanwhile, pioneering search-based program repair techniques (e.g., GenProg [38]) tentatively change program elements based on certain rules (e.g., deleting/changing/adding program elements) and use the original test suite as the oracle to validate the generated patches. Please refer to the recent surveys on automated software debugging for more details [50, 76]. To date, unfortunately, although debugging has been extensively studied and even has drawn attention from industry (e.g., FaceBook [47, 66] and Fujitsu [65]), *we still lack practical automated debugging techniques*: (1) existing fault localization techniques have been shown to have limited effectiveness in practice [56, 77]; (2) existing program repair techniques can only fix a small ratio of real bugs [20, 29, 73] or specific types of bugs [47].

In this work, we aim to revisit the connection between program repair and fault localization for more powerful debugging. We observe that the current existing connection between fault localization and program repair is that program repair techniques usually use off-the-shelf fault localization techniques to identify potential buggy locations for patching, e.g., the Ochiai [5] SBFL technique is leveraged in many recent program repair techniques [20, 29, 73]. In this work, we propose a new *unified debugging* approach to unify program repair and fault localization in the reversed way, and explore the following question, *can program repair in turn help with fault localization?* Our basic insight is that the patch execution information during program repair can provide useful feedbacks and guidelines for powerful fault localization. For example, if a patch passes some originally failing test(s), the patched location is very likely to be closely related to the real buggy location (e.g., sharing the same method or even same line), since otherwise the patch cannot mute the bug impacts to pass the failing test(s). Based on this insight, we designed ProFL (**P**rogram **R**epair for **F**ault **L**ocalization), a simplistic feedback-driven fault localization approach that leverages patch-execution information from state-of-the-art PraPR [20] repair tool for rearranging fault localization results computed by off-the-shelf fault localization techniques. Note that even state-of-the-art program repair techniques can only fix a small ratio of real bugs (i.e., <20% for Defects4J [20, 29, 73]) fully automatically and were simply aborted for the vast majority of unfixed bugs, while our approach extends the application scope of program repair to all possible bugs – *program repair techniques can also provide useful fault localization information to help with manual repair even for the bugs that are hard to fix automatically*.

We have evaluated our ProFL on the Defects4J (V1.2.0) benchmark, which includes 395 real-world bugs from six open-source Java projects and has been widely used for evaluating both fault localization and program repair techniques [20, 29, 40, 67, 73]. Our experimental results show that ProFL can localize 161 bugs within Top-1, while state-of-the-art spectrum and mutation based fault localization techniques can at most localize 117 bugs within Top-1. We further investigate the impacts of various experimental configurations: (1) we investigate the finer-grained patch categorizations

and observe that they do not have clear impact on ProFL; (2) we investigate the impact of different off-the-shelf SBFL formulae used in ProFL, and observe that ProFL consistently outperforms traditional SBFL regardless of the used formulae; (3) we replace the repair feedback information with traditional mutation feedback information in ProFL (since they both record the impacts of certain changes to test outcomes), and observe that ProFL still localizes 141 bugs within Top-1, significantly outperforming state-of-the-art SBFL and MBFL; (4) we feed ProFL with only *partial* patch-execution information (since the test execution will be aborted for a patch as soon as it gets falsified by some test for the sake of efficiency in practical program repair scenario), and observe that, surprisingly, ProFL using such partial information can reduce the execution overhead by 26.17X with no clear effectiveness drop; (5) we also apply ProFL on a newer version of Defects4J, Defects4J (V1.4.0) [22], and observe that ProFL performs consistently. In addition, we further observe that ProFL can even significantly boost state-of-the-art fault localization via both unsupervised [82, 83] and supervised [39] learning, localizing 185 and 216.80 bugs within Top-1, the best fault localization results via unsupervised/supervised learning to our knowledge.

This paper makes the following contributions:

- This paper opens a new unified debugging dimension for improving fault localization via off-the-shelf state-of-the-art program repair techniques, and also extends the application scope of program repair techniques to all possible bugs (not only the bugs that can be directly automatically fixed).
- The proposed approach, ProFL, has been implemented as a fully automated Maven plugin<sup>1</sup> for automated feedback-driven fault localization based on the patch executions of PraPR, a state-of-the-art program repair technique.
- We have performed an extensive study of the proposed approach on the widely used Defects4J benchmarks, and demonstrated the effectiveness, efficiency, robustness, and general applicability of the proposed approach.
- ProFL plugin has been deployed in an international IT company with over 1 billion global users. We also conducted a real-world industry case study within the company.

## 2 BACKGROUND AND RELATED WORK

**Fault Localization** [5, 9, 14, 23–25, 28, 30, 42, 51, 57, 62–64, 80, 81] aims to precisely diagnose potential buggy locations to facilitate manual bug fixing. The most widely studied *spectrum-based fault localization* (SBFL) techniques usually apply statistical analysis (e.g., Tarantula [30], Ochiai [5], and Ample [14]) or learning techniques [9, 62–64] to the execution traces of both passed and failed tests to identify the most suspicious code elements (e.g., statements/methods). The insight is that code elements primarily executed by failed tests are more suspicious than the elements primarily executed by passed tests. However, a code element executed by a failed test does not necessarily indicate that the element has impact on the test execution and has caused the test failure. To bridge the gap between coverage and impact information, researchers proposed *mutation-based fault localization* (MBFL) [51, 54, 55, 81], which injects changes to each code element (based on mutation testing [17, 26]) to check its impact on the test outcomes. MBFL has

<sup>1</sup><https://github.com/yilinglou/proFL>

been applied to both general bugs (pioneered by Metallaxis [54, 55]) and regression bugs (pioneered by FiFL [81]). ProFL shares similar insight with MBFL in that program changes can help determine the impact of code elements on test failures. However, ProFL utilizes program repair information that aims to fix software bugs to *pass* more tests rather than mutation testing that was originally proposed to create new artificial bugs to *fail* more tests; ProFL also embodies a new feedback-driven fault localization approach. Besides SBFL and MBFL, researchers have proposed to utilize various other information for fault localization (such as program slicing [80], predicate switching [84], code complexity [67], program invariant [7] information, and bug reports [37]), and have also utilized supervised learning to incorporate such different feature dimensions for fault localization [39, 40, 79]. However, the effectiveness of supervised-learning-based fault localization techniques may largely depend on the training sets, which may not always be available. In this work, we aim to explore a new direction for simplistic fault localization without supervised learning, i.e., leveraging patch-execution information (from program repair) for powerful fault localization.

**Automated Program Repair (APR) techniques** [12, 15, 19, 21, 27, 44–46, 48–50, 53, 58, 71, 78] aim to directly fix software bugs with minimal human intervention via synthesizing *genuine* patches (i.e., the patches semantically equivalent to developer patches). Therefore, despite a young research area, APR has been extensively studied in the literature. Various techniques have been proposed to directly modify program code representations based on different rules/strategies, and then use tests as the oracle to validate each generated candidate patch to find *plausible* patches (i.e., the patches passing all tests/checks) [12, 15, 21, 45, 49, 53, 58, 78]. Note that not all plausible patches are genuine patches; thus existing APR techniques all rely on manual inspection to find the final genuine patches among all plausible ones. *Search-based* APR techniques assume that most bugs could be solved by searching through all the potential candidate patches based on certain patching rules (i.e., program-fixing templates) [16, 29, 38, 73]. Alternatively, *semantics-based* techniques use deeper semantical analyses (including symbolic execution [13, 33]) to synthesize program conditions, or even more complex code snippets, that can pass all the tests [49, 53, 78]. Recently, search-based APR has been extensively studied due to its scalability on real-world systems, e.g., the most recent PraPR technique has been reported to produce genuine patches for 43 real bugs from Defects4J [31]. Despite the success of recent advanced APR techniques, even the most recent program repair technique can only fix a small ratio (i.e., <20% for Defects4J) of real bugs [20, 29, 73] or specific types of bugs [47].

In this work, we aim to leverage program repair results to help with fault localization. More specifically, we design, ProFL, a simplistic feedback-driven fault localization approach guided by patch-execution results (from program repair). Note that Ghanbari et al. [20] and Timperley et al. [69] have considered that plausible patches may potentially help localize bugs. However, Ghanbari et al. did not present a systematic fault localization approach working for all possible bugs, while Timperley et al. showed that this direction is ineffective. In contrast, we present the first systematic fault localization approach driven by state-of-the-art program repair, perform the first extensive study under various settings and on a large number of real-world bugs, and show for the first time that state-of-the-art

```

Class: org.apache.commons.math.analysis.solvers.BoundingNthOrderBrentSolver
Method: protected double doSolve()
Developer patch:
233: if (agingA >= MAXIMAL_AGING) {
234: // ...
235: - targetY = -REDUCTION_FACTOR * yB;
236: + final int p = agingA - MAXIMAL_AGING;
237: + final double weightA = (1 << p) - 1;
238: + final double weightB = p + 1;
239: + targetY = (weightA * yA - weightB * REDUCTION_FACTOR * yB)
/ (weightA + weightB);
} else if (agingB >= MAXIMAL_AGING) {
240: - targetY = -REDUCTION_FACTOR * yA;
241: // ...
242: + final int p = agingB - MAXIMAL_AGING;
243: + final double weightA = p + 1;
244: + final double weightB = (1 << p) - 1;
245: + targetY = (weightB * yB - weightA * REDUCTION_FACTOR * yA)
/ (weightA + weightB);

Patch P4, generated by PraPR:
260: - if (signChangeIndex - start >= end - signChangeIndex) {
261: + if (MAXIMAL_AGING - start >= end - signChangeIndex) {
262:     ++start;
263: } else {
264:     --end;
}

Patch P5, generated by PraPR:
317: - x[signChangeIndex] = nextX;
318: + x[agingA] = nextX;
319: System.arraycopy(y, signChangeIndex, y, signChangeIndex +
1, nbPoints - signChangeIndex);
y[signChangeIndex] = nextY;

```

Figure 1: Developer and generated patches for Math-40

```

Class: com.google.javascript.jscomp.NodeUtil
Method: static boolean functionCallHasSideEffects
Developer patch:
958: + if (nameNode.getFirstChild().getType() == Token.NAME) {
959: + String namespaceName = nameNode.getFirstChild().getString()
();
960: + if (namespaceName.equals("Math")) {
961: +     return false;
962: + }
963: + }

Patch P10, generated by PraPR:
933: - if (callNode.isNoSideEffectsCall()) {
933: + if (callNode.hasChildren()) {
934:     return false;
935: }

```

Figure 2: Developer and generated patches for Closure-61

*program repair can substantially boost state-of-the-art fault localization.* Feedback-driven fault localization techniques have also been investigated before [41, 43]. However, existing feedback-driven fault localization techniques usually require manual inspection to guide debugging. In contrast, we present a fully automated feedback-driven fault localization, i.e., ProFL utilizes patch-execution results as feedback to enable powerful automated fault localization.

## 3 MOTIVATING EXAMPLES

In this section, we present two real-world bug examples to show the potential benefits of program repair for fault localization.

### 3.1 Example 1: Math-40

We use Math-40 from Defects4J (V1.2.0) [31], a widely used collection of real-world Java bugs, as our first example. Math-40 denotes the 40th buggy version of Apache Commons Math project [6] from

**Table 1: Five top-ranked methods from Math-40**

EID	Method Signature	SBFL	PID	#F (1)	#P (3177)
$e_1$	incrementEvaluationCount()	0.57	$\mathcal{P}_1$	1	3170
$e_2$	BracketingNthOrderBrentSolver(Number)	0.33	$\mathcal{P}_2$	1	3172
$e_3$	BracketingNthOrderBrentSolver(double, ...)	0.28	$\mathcal{P}_3$	1	3177
$e_4^*$	doSolve()	0.27	$\mathcal{P}_4$	0	3177
$e_5$	guessX(double[], ...)	0.20	$\mathcal{P}_5$	0	3169
			$\mathcal{P}_6$	0	3176

Defects4J (V1.2.0). The bug is located in a single method of the project (method `doSolve` of class `BracketingNthOrderBrentSolver`).

We attempted to improve the effectiveness of traditional SBFL based on Ochiai formula [5], which has been widely recognized as one of the most effective SBFL formulae [40, 57, 82]. Inspired by prior work [67], we used the *aggregation* strategy to aggregate the maximum suspiciousness values from statements to methods. Even with this improvement in place, Ochiai still cannot rank the buggy method in the top, and instead ranks the buggy method in the 4th place (with a suspiciousness value of 0.27). The reason is that traditional SBFL captures only coverage information and does not consider the actual impacts of code elements on test behaviors.

In an attempt to fix the bug, we further applied state-of-the-art APR technique, PraPR [20], on the bug. However, since fixing the bug requires multiple lines of asymmetric edits, the genuine patch is beyond the reach of PraPR and virtually other existing APR techniques as well. Analyzing the generated patches and their execution results, however, gives some insights on the positive effects that an APR technique might have on fault localization.

Among a large number of methods in Math-40, Table 1 lists the Top-5 most suspicious methods based on Ochiai. Each row corresponds to a method, with the highlighted one corresponding to the actual buggy method (i.e., `doSolve`). Column “EID” assigns an identifier for each method. Column “SBFL” reports SBFL suspiciousness values for each method, and “PID” assigns an identifier for each patch generated by PraPR that targets the method. Column “#F”/#P” reports the number of originally failing/passing tests that still fail/pass on each generated patch. The numbers within the parentheses in the table head are the number of failing/passing tests on the original buggy program. We also present the details of the developer patch for the bug and two patches generated by PraPR on the buggy method in Figure 1. From the table, we observe that  $\mathcal{P}_4$  is a plausible patch, meaning that it passes all of the available tests but it might be not a genuine fix;  $\mathcal{P}_5$  passes originally failing tests, while fails to pass 8 originally passing tests.

Several observations can be made at this point: First, whether the originally failing tests pass or not on a patch, can help distinguish the buggy methods from some correct methods. For example, for  $e_1$ ,  $e_2$  and  $e_3$ , the originally failing test remains failing on all of their patches, while for the buggy method  $e_4$ , the originally failing test becomes passing on both its patches. Second, whether the originally passing tests fail or not, can also help separate the buggy methods from some correct methods, e.g.,  $\mathcal{P}_4$  for the buggy method  $e_4$  does not fail any originally passing tests while the patch for the correct method  $e_5$  still fails some originally passing tests. Lastly, the detailed number of tests affected by the patches may not matter much. For example, for the correct method  $e_5$ , its patch only fails one originally passing test, but for the buggy method  $e_4$ , patch  $\mathcal{P}_5$  makes even more (i.e., 8) originally passing tests fail.

**Table 2: Five top-ranked methods from Closure-61**

EID	Method Signature	SBFL	PID	#F (3)	#P (7082)
$e_1$	<code>toString()</code>	0.34	$\mathcal{P}_7$	3	7079
$e_2$	<code>getSortedPropTypes()</code>	0.33	$\mathcal{P}_8$	3	6981
$e_3$	<code>toString(StringBuilder, ...)</code>	0.27	$\mathcal{P}_9$	3	7042
$e_4^*$	<code>functionCallHasSideEffects(Node, ...)</code>	0.18	$\mathcal{P}_{10}$	1	6681
$e_5$	<code>nodeTypeMayHaveSideEffects(Node, ...)</code>	0.09	$\mathcal{P}_{11}$	1	6766

### 3.2 Example 2: Closure-61

We further looked into Closure-61, another real-world buggy project from Defects4J (V1.2.0), but for which PraPR is even *unable* to generate any plausible patch. Similar with the first example, we present the Ochiai and PraPR results for the Top-5 methods in Table 2.

Based on Table 2, we observe that even the non-plausible noisy patch  $\mathcal{P}_{10}$  is related to the buggy methods. The patches targeting method `getSortedPropTypes` and the two overloading methods of `toString` (which have higher suspiciousness values than that of the buggy method `functionCallHasSideEffects`) cannot generate any patch that can pass any of the originally failing tests. In addition, the fact that the number of passed tests which now fail in the patches of the buggy method are much larger than that for the correct method `nodeTypeMayHaveSideEffects` further confirms our observation above that, the detailed impacted test number does not matter much with the judgement of the correctness of a method.

Based on the above two examples, we have following implications to utilize the patch execution results to improve the original SBFL: (1) the patches (no matter plausible or not) positively impacting some failed test(s) may indicate the actual buggy locations and should be favored; (2) the patches negatively impacting some passed test(s) may help exclude some correct code locations and should be unfavored; (3) the detailed number of the impacted tests does not matter much for fault localization. Therefore, we categorize all the patches into four different basic groups based on whether they impact originally passed/failed tests to help with fault localization, details shown in Section 4.

## 4 APPROACH

### 4.1 Preliminaries

In order to help the readers better understand the terms used throughout this paper, in what follows, we attempt to define a number of key notions more precisely.

**Definition 4.1 (Candidate Patch).** Given the original program  $\mathcal{P}_o$ , a candidate patch  $\mathcal{P}$  can be created by modifying one or more program elements within  $\mathcal{P}_o$ . The set of all candidate patches generated for the program is denoted by  $\mathbb{P}$ .

In this paper, we focus on APR that conducts only *first-order* program transformations, which only change one program element in each patch, such as PraPR [20]. Note that our approach is general and can also be applied to other APR techniques in theory, even including the ones applying *high-order* program transformations.

**Definition 4.2 (Patch Execution Matrix).** Given a program  $\mathcal{P}_o$ , its test suite  $\mathcal{T}$ , and its corresponding set of all candidate patches  $\mathbb{P}$ , the patch execution matrix,  $\mathbb{M}$ , is defined as the execution results of all patches in  $\mathbb{P}$  on all tests in  $\mathcal{T}$ . Each matrix cell result,  $\mathbb{M}[\mathcal{P}, t]$ , represents the execution results of test  $t \in \mathcal{T}$  on patch  $\mathcal{P} \in \mathbb{P}$ , and

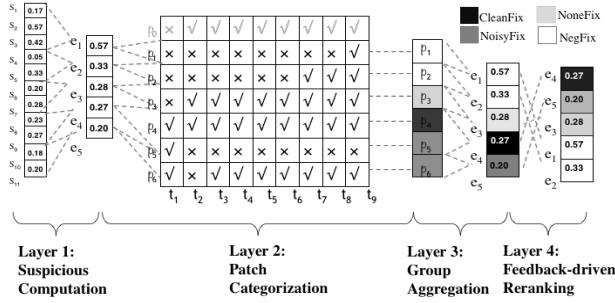


Figure 3: Overview of ProFL

can have the following possible values,  $\{\checkmark, \times, \circlearrowleft\}$ , which represent *failed*, *passed*, and *unknown yet*.

Note that for the ease of presentation, we also include the original program execution results in  $\mathbb{M}$ , i.e.,  $\mathbb{M}[\mathcal{P}_o, t]$  denotes the execution results of test  $t$  on the original program  $\mathcal{P}_o$ .

Based on the above definitions, we can now categorize candidate patches based on the insights obtained from motivating examples:

**Definition 4.3 (Clean-Fix Patch).** A patch  $\mathcal{P}$  is called a Clean-Fix Patch, i.e.,  $\mathbb{G}[\mathcal{P}] = \text{CleanFix}$ , if it passes some originally failing tests while does not fail any originally passing tests, i.e.,  $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \times \wedge \mathbb{M}[\mathcal{P}, t] = \checkmark$ , and  $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \checkmark \wedge \mathbb{M}[\mathcal{P}, t] = \times$ .

Note that  $\mathbb{G}[\mathcal{P}]$  returns the category group for each patch  $\mathcal{P}$ .

**Definition 4.4 (Noisy-Fix Patch).** A patch  $\mathcal{P}$  is called a Noisy-Fix Patch, i.e.,  $\mathbb{G}[\mathcal{P}] = \text{NoisyFix}$ , if it passes some originally failing tests but also fails on some originally passing tests, i.e.,  $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \times \wedge \mathbb{M}[\mathcal{P}, t] = \checkmark$ , and  $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \checkmark \wedge \mathbb{M}[\mathcal{P}, t] = \times$ .

**Definition 4.5 (None-Fix Patch).** A patch  $\mathcal{P}$  is called a None-Fix Patch, i.e.,  $\mathbb{G}[\mathcal{P}] = \text{NoneFix}$ , if it does not impact any originally failing or passing tests. More precisely,  $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \times \wedge \mathbb{M}[\mathcal{P}, t] = \checkmark$ , and  $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \checkmark \wedge \mathbb{M}[\mathcal{P}, t] = \times$ .

**Definition 4.6 (Negative-Fix Patch).** A patch  $\mathcal{P}$  is called a Negative-Fix Patch, i.e.,  $\mathbb{G}[\mathcal{P}] = \text{NegFix}$ , if it does not pass any originally failing test while fails some originally passing tests, i.e.,  $\nexists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \times \wedge \mathbb{M}[\mathcal{P}, t] = \checkmark$ , and  $\exists t \in \mathcal{T}, \mathbb{M}[\mathcal{P}_o, t] = \checkmark \wedge \mathbb{M}[\mathcal{P}, t] = \times$ .

Based on our insights obtained from the motivating example, the ranking of different patch groups is: CleanFix > NoisyFix > NoneFix > NegFix. Note that in Section 4.3, we will discuss more patch categorization variants besides such default patch categorization to further study their impacts on ProFL.

## 4.2 Basic ProFL

The overview of ProFL is shown in Figure 3. According to the figure, ProFL consists of four different layers. The input for ProFL is the actual buggy program under test and the original failing test suite, and the final output is a refined ranking of the program elements based on the initial suspiciousness calculation. In the first layer, ProFL conducts naive SBFL formulae (e.g., Ochiai [5]) at the statement level, and then performs *suspiciousness aggregation* [67] to calculate the initial suspiciousness value for each program element. Note that besides such default initial suspiciousness computation, ProFL is generic and can leverage the suspiciousness values computed

by any other advanced fault localization technique in this layer (such as the PageRank-based fault localization [82]). In the second layer, ProFL collects the patch execution matrix along the program repair process for the program under test, and categorizes each patch into different groups based on Section 4.1. In the third layer, for each element, ProFL maps the group information of its corresponding patches to itself via *group aggregation*. In the last layer, ProFL finally reranks all the program elements via considering their suspiciousness and group information in tandem.

We next explain each layer in detail with our first motivation example. Since the number of tests and patches are really huge, due to space limitation, we only include the tests and patches that are essential for the ranking results of the elements. After reduction, we consider the six patches ( $\mathcal{P}_1$  to  $\mathcal{P}_6$ ) and the 9 tests whose statuses changed on these patches (denoted as  $t_1$  to  $t_9$ ). Based on Definition 4.2, we present  $\mathbb{M}$  in Figure 3. The first row stands for  $\mathbb{M}[\mathcal{P}_o, \mathcal{T}]$ , the execution results of  $\mathcal{T}$  on the original buggy program  $\mathcal{P}_o$ , and from the second row, each row represents  $\mathbb{M}[\mathcal{P}, \mathcal{T}]$ , the execution results of each patch  $\mathcal{P}$  as shown in Table 1 on  $\mathcal{T}$ .

**4.2.1 Layer 1: Suspicious Computation.** Given the original program statements, e.g.,  $\mathcal{S} = [s_1, s_2, \dots, s_n]$ , we directly apply an off-the-shelf spectrum-based fault localization technique (e.g., the default Ochiai [5]) to compute the suspiciousness for each statement, e.g.,  $\mathbb{S}[s_j]$  for statement  $s_j$ . Then, the proposed approach applies *suspiciousness aggregation* [67] to compute the element suspiciousness values at the desired level (e.g., method level in this work) since prior work has shown that suspicious aggregation can significantly improve fault localization results [11, 67]. Given the initial list of  $\mathcal{E} = [e_1, e_2, \dots, e_m]$ , for each  $e_i \in \mathcal{E}$ , suspiciousness aggregation computes its suspiciousness as  $\mathbb{S}[e_i] = \text{Max}_{s_j \in e_i} \mathbb{S}[s_j]$ , i.e., the highest suspiciousness value for all statements within a program element is computed as the suspiciousness value for the element.

For our first motivation example, after suspicious aggregation, for the five elements,  $\mathbb{S}[e_1, e_2, e_3, e_4, e_5] = [0.57, 0.33, 0.28, 0.27, 0.20]$ .

**4.2.2 Layer 2: Patch Categorization.** In this layer, ProFL automatically invokes off-the-shelf program repair engines (PraPR [20] for this work) to try various patching opportunities and record the detailed patch-execution matrix,  $\mathbb{M}$ . Then, based on the resulting  $\mathbb{M}$ , ProFL automatically categorizes each patch into different groups. Given program element  $e$  and all the patches generated for the program,  $\mathbb{P}$ , the patches occurring on  $e$  can be denoted as  $\mathbb{P}[e]$ . Then, based on Definitions 4.3 to 4.6, each patch within  $\mathbb{P}[e]$  for each element  $e$  can be categorized into one of the four following groups, {CleanFix, NoisyFix, NoneFix, NegFix}. Recall that  $\mathbb{G}[\mathcal{P}]$  represents the group information for  $\mathcal{P}$ , e.g.,  $\mathbb{G}[\mathcal{P}] = \text{CleanFix}$  denotes that  $\mathcal{P}$  is a clean-fix patch.

For the example, the group of each patch in the motivation example is as follows:  $\mathbb{G}[\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4, \mathcal{P}_5, \mathcal{P}_6] = [\text{NegFix}, \text{NegFix}, \text{NoneFix}, \text{CleanFix}, \text{NoisyFix}, \text{NoisyFix}]$

**4.2.3 Layer 3: Group Aggregation.** For each program element  $e$ , we utilize its corresponding patch group information to determine its own group information. Recall that the ranking of different patch groups is: CleanFix > NoisyFix > NoneFix > NegFix. Then, the group information for a program element can be determined by the best group information of all patches occurring on the program

element. Therefore, we present the following rules for determining the group information for each  $e$ :

$$\mathbb{G}[e] = \begin{cases} \text{CleanFix} & \text{if } \exists \mathcal{P}, \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \text{CleanFix} \\ \text{NoisyFix} & \text{else if } \exists \mathcal{P}, \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \text{NoisyFix} \\ \text{NoneFix} & \text{else if } \exists \mathcal{P}, \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \text{NoneFix} \\ \text{NegFix} & \text{else if } \exists \mathcal{P}, \mathcal{P} \in \mathbb{P}[e] \wedge \mathbb{G}[\mathcal{P}] = \text{NegFix} \end{cases} \quad (1)$$

Shown in Equation 1, element  $e$  is within Group CleanFix whenever there is any patch  $\mathcal{P}$  within  $e$  such that  $\mathcal{P}$  is a clean-fix patch; otherwise, it is within Group NoisyFix whenever there is any patch  $\mathcal{P}$  within  $e$  such that  $\mathcal{P}$  is a noisy-fix patch.

After group aggregation, the group of each program element (i.e., method) in the motivation example is  $\mathbb{G}[e_1, e_2, e_3, e_4, e_5] = [\text{NegFix}, \text{NegFix}, \text{NoneFix}, \text{CleanFix}, \text{NoisyFix}]$ .

**4.2.4 Layer 4: Feedback-Driven Reranking.** In this last layer, we compute the final ranked list of elements based on the aggregated suspiciousness values and groups. All the program elements will be first clustered into different groups with Group CleanFix ranked first and Group NegFix ranked last. Then, within each group, the initial SBFL (or other fault localization techniques) suspiciousness values will be used to rank the program elements. Assume we use  $\mathbb{R}[e_1, e_2]$  to denote the total-order ranking between any two program elements, it can be formally defined as:

$$\mathbb{R}[e_1, e_2] = \begin{cases} e_1 \geq e_2 & \text{if } \mathbb{G}[e_1] > \mathbb{G}[e_2] \text{ or} \\ & \mathbb{G}[e_1] = \mathbb{G}[e_2] \wedge \mathbb{S}[e_1] \geq \mathbb{S}[e_2] \\ e_2 \geq e_1 & \text{if } \mathbb{G}[e_2] > \mathbb{G}[e_1] \text{ or} \\ & \mathbb{G}[e_1] = \mathbb{G}[e_2] \wedge \mathbb{S}[e_2] \geq \mathbb{S}[e_1] \end{cases} \quad (2)$$

That is,  $e_1$  is ranked higher or equivalent to  $e_2$  only when (i)  $e_1$  is within a higher-ranked group, or (ii)  $e_1$  is within the same group as  $e_2$  but has a higher or equivalent suspicious value compared to  $e_2$ . Therefore, the final ranking of our example is:  $e_4 \geq e_5 \geq e_3 \geq e_1 \geq e_2$ , ranking the buggy method  $e_4$  at the first place.

### 4.3 Variants of ProFL

Taking the approach above as the basic version of ProFL, there can be many variants of ProFL, which are discussed as follows.

**Finer-grained Patch Categorization.** Previous work [20] found that plausible patches are often coupled tightly with buggy elements, which actually is a subset of CleanFix defined in our work. Inspired by this finding, we further extend ProFL with finer-grained patch categorization rules, which respectively divide CleanFix and NoisyFix into two finer categories according to the criterion whether all failed tests are impacted. We use Figure 4 to show the relation between the four finer-grained patch categories and the four basic categories. Considering the finer categories, we further extend the group aggregation strategies in the third layer of ProFL accordingly as shown in Table 3 to study the impact of further splitting CleanFix and NoisyFix categories. For example,  $R_1$  (ranking CleanPartFix below CleanAllFix) and  $R_2$  (ranking CleanAllFix below CleanPartFix) study the two different ways for splitting CleanFix.

**SBFL Formulae.** The elements are reranked in the last layer based on their aggregated suspiciousness values and groups. In theory, ProFL is not specific for any particular way to calculate the aggregated suspiciousness value. Therefore, besides our default Ochiai [5]

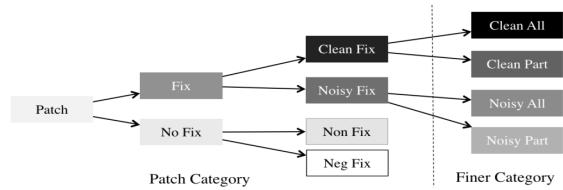


Figure 4: Patch categorization tree

Table 3: Finer-grained patch categorization rules

ID	Extended Categorization Aggregation Rules
$R_1$	CleanAllFix > CleanPartFix > NoisyFix > NoneFix > NegFix
$R_2$	CleanPartFix > CleanAllFix > NoisyFix > NoneFix > NegFix
$R_3$	CleanFix > NoisyAllFix > NoisyPartFix > NoneFix > NegFix
$R_4$	CleanFix > NoisyPartFix > NoisyAllFix > NoneFix > NegFix

formula, all the other formulae in SBFL can be adopted in ProFL. We study all the 34 SBFL formulae considered in prior work [40, 67]. The impact of these formulae on ProFL would be studied later.

**Feedback Sources.** Generally speaking, not only the patch execution results can be the feedback of our approach, any other execution results correlated with program modifications can serve as the feedback sources, e.g., mutation testing [26]. For example, a mutant and a patch are both modifications on the program, thus ProFL can directly be applied with the mutation information as feedback. However, mutation testing often includes simple syntax modifications that were originally proposed to simulate software bugs to *fail* more tests, while program repair often includes more (advanced) modifications that aim to *pass* more tests to fix software bugs. Therefore, although it is feasible to use mutation information as the feedback source of our approach, the effectiveness remains unknown, which would be studied.

**Partial Execution Matrix.** During program repair, usually the execution for a patch would terminate as soon as one test fails, which is the common practice to save the time cost. In this scenario, only partial execution results are accessible. In the previous sections,  $\mathbb{M}$  is considered as complete (as traditional MBFL also requires full mutant execution matrices), which we denote as *full* matrix,  $\mathbb{M}_f$ , while in this section, we discuss the case where  $\mathbb{M}$  is considered as incomplete in APR practice, which we call a *partial* matrix,  $\mathbb{M}_p$ . Recall Definition 4.2, different from  $\mathbb{M}_f$ , cells in  $\mathbb{M}_p$  can be  $\textcircled{O}$  besides  $\checkmark$  and  $\texttimes$ . E.g., when  $t$  is not executed on  $\mathcal{P}$ ,  $\mathbb{M}_p[\mathcal{P}, t] = \textcircled{O}$ .

In the motivation example, during the patch execution, if  $\mathcal{T}$  is executed in the order from  $t_1$  to  $t_9$ , and one failed test would stop execution for each patch immediately,  $\mathbb{M}_p$  is as follows.

$$\mathbb{M}_p = \left[ \begin{array}{ccccccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \\ \mathcal{P}_0 & \texttimes & \checkmark \\ \mathcal{P}_1 & \texttimes & \textcircled{O} \\ \mathcal{P}_2 & \texttimes & \textcircled{O} \\ \mathcal{P}_3 & \texttimes & \textcircled{O} \\ \mathcal{P}_4 & \checkmark \\ \mathcal{P}_5 & \checkmark & \texttimes & \textcircled{O} \\ \mathcal{P}_6 & \checkmark & \texttimes & \textcircled{O} \end{array} \right] \quad (3)$$

In the scenario where only partial matrix is accessible, we can find there are many unknown results. Interestingly, in this example, we find the final ranking does not change at all even with a partial

**Table 4: Benchmark statistics**

ID	Name	#Bug	#Test	LoC
Lang	Apache commons-lang	65	2,245	22K
Math	Apache commons-math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
Chart	JFreeChart	26	2,205	96K
Closure	Google Closure compiler	133	7,927	90K
Mockito	Mockito framework	38	1,366	23K
<b>Defects4J (V1.2.0)</b>		<b>395</b>	<b>21,475</b>	<b>344K</b>
Cli	Apcache commons-cli	24	409	4K
Codec	Apache commons-codec	22	883	10K
Csv	Apache commons-csv	12	319	2K
JXPath	Apache commons-jxpath	14	411	21K
Gson	Google GSON	16	N/A	12K
Guava	Google Guava	9	1,701,947	420K
Core	Jackson JSON processor	13	867	31K
Databind	Jackson data bindings	39	1,742	71K
Xml	Jackson XML extensions	5	177	6K
Jsoup	Jsoup HTML parser	63	681	14K
<b>Defects4J (V1.4.0)</b>		<b>587</b>	<b>26,964</b>	<b>503K</b>

matrix as input. For the patches  $\mathcal{P}_3$ ,  $\mathcal{P}_4$ ,  $\mathcal{P}_5$  and  $\mathcal{P}_6$ , their patch categorization does not change at all. For example, since the failed tests are executed first, when  $\mathcal{P}_5$  stops its execution, its execution result is that one failed test passes now and one passed test fails now, and thus  $\mathcal{P}_5$  is still categorized into NoisyFix. For  $\mathcal{P}_1$  and  $\mathcal{P}_2$ , although their patch categorization changes from NegFix to NoneFix, it does not impact the final ranking results. The example indicates the insensitivity of ProFL to partial matrix, and the categorization design is the main reason for it. We would further confirm this observation in the detailed experimental studies.

## 5 EXPERIMENT SET UP

### 5.1 Research Questions

In our study, we investigate the following research questions:

- **RQ1:** How does the basic ProFL perform compared with state-of-the-art SBFL and MBFL techniques?
- **RQ2:** How do different configurations impact ProFL?
  - **RQ2a:** What is the impact of finer patch categorization?
  - **RQ2b:** What is the impact of the used SBFL formula?
  - **RQ2c:** What is the impact of the feedback source used?
  - **RQ2d:** What is the impact of partial execution matrix?
  - **RQ2e:** What is the impact of the used benchmark suite?
- **RQ3:** Can ProFL further boost state-of-the-art unsupervised- and supervised-learning-based fault localization?

Besides the research questions, we also conduct a case study in an international IT company with over 700M Monthly Active Users.

### 5.2 Benchmark

We conduct our study on all bugs from the Defects4J benchmark [31], which has been widely used in prior fault-localization work [39, 40, 57, 67, 82]. Defects4J is a collection of reproducible real bugs with a supporting infrastructure. To our knowledge, all the fault localization studies evaluated on Defects4J use the original version Defects4J (V1.2.0). Recently, an extended version, Defects4J (V1.4.0),

which includes more real-world bugs, has been released [22]. Therefore, we further perform the first fault localization study on Defects4J (V1.4.0) to reduce the threats to external validity.

In Table 4, Column “ID” presents the subject IDs. Columns “Name” and “#Bugs” present the full name and the number of bugs for each project. Columns “Loc” and “#Test” list the line-of-code information and the number of tests for the HEAD version of each project. Note that the two projects highlighted in gray are excluded due to build/test framework incompatibility with PraPR [20]. In total, our study is performed on all 395 bugs from Defects4J (V1.2.0) and 192 additional bugs from Defects4J (V1.4.0).

### 5.3 Independent Variables

**Evaluated Techniques:** We compare ProFL with the following state-of-the-art SBFL and MBFL techniques: **(a) Spectrum-based (SBFL):** we consider traditional SBFL with suspiciousness aggregation strategy to aggregate suspiciousness values from statements to methods, which has been shown to be more effective than naive SBFL in previous work [11, 67]. **(b) Mutation-based (MBFL):** we consider two representative MBFL techniques, MUSE [51] and Metallaxis [55]. **(c) Hybrid of SBFL and MBFL (MCBFL):** we also consider the recent MCBFL [57], which represents state-of-the-art hybrid spectrum- and mutation-based fault localization. Furthermore, we include state-of-the-art learning-based fault localization techniques: **(a) Unsupervised:** we consider state-of-the-art PRFL [82] and PRFL<sub>MA</sub> [83] (which further improves PRFL via suspiciousness aggregation) that aim to boost SBFL with the unsupervised PageRank algorithm. **(b) Supervised:** we consider state-of-the-art supervised-learning-based fault localization, DeepFL [39], which outperforms all other learning-based fault localization [40, 67, 79]. Note that, SBFL and Metallaxis can adopt different SBFL formulae, and we by default uniformly use Ochiai [5] since it has been demonstrated to perform the best for both SBFL and MBFL [40, 57, 82].

**Experimental Configurations:** We explore the following configurations to study ProFL: **(a) Finer ProFL Categorization:** in RQ2a, we study the four extended categorization aggregation rules based on the finer patch categories as listed in Table 3. **(b) Studied SBFL Formulae:** in RQ2b, we implement all the 34 SBFL formulae considered in prior work [40, 67] to study the impact of initial formulae. **(c) Feedback Sources:** besides the patch execution results of program repair, mutation testing results can also be used as the feedback sources of ProFL. Thus, we study the impact of these two feedback sources in RQ2c. **(d) Partial Execution Matrix:** we collect partial execution matrices in three common test-execution orderings: **(i)**  $O_1$ : the default order in original test suite; **(ii)**  $O_2$ : running originally-failed tests first and then originally-passing tests, which is also the common practice in program repair to save the time cost; **(iii)**  $O_3$ : running originally-passing tests first and then originally-failed tests. The partial matrices collected by these orders are denoted as  $M_p^{(O_1)}$ ,  $M_p^{(O_2)}$  and  $M_p^{(O_3)}$  respectively. We investigate the impacts of different partial execution matrices used in RQ2d. **(e) Used Benchmarks:** we evaluate ProFL in two benchmarks, Defects4J (V1.2.0) and Defects4J (V1.4.0) in RQ2e.

## 5.4 Dependent Variables and Metrics

In this work, we perform fault localization at the method level following recent fault localization work [7, 39, 40, 67, 82], because the class level has been shown to be too coarse-grained while the statement level is too fine-grained to keep useful context information [35, 56]. We use the following widely used metrics [39, 40]:

**Recall at Top-N:** Top-N computes the number of bugs with at least one buggy element localized in the Top-N positions of the ranked list. As suggested by prior work [56], usually, programmers only inspect a few buggy elements in the top of the given ranked list, e.g., 73.58% developers only inspect Top-5 elements [35]. Therefore, following prior work [39, 40, 82, 85], we use Top-N ( $N=1, 3, 5$ ).

**Mean First Rank (MFR):** For each subject, MFR computes the mean of the first relevant buggy element's rank for all its bugs, because the localization of the first buggy element for each bug can be quite crucial for localizing all buggy elements.

**Mean Average Rank (MAR):** We first compute the average ranking of all the buggy elements for each bug. Then, MAR of each subject is the mean of such average ranking of all its bugs. MAR emphasizes the precise ranking of all buggy elements, especially for the bugs with multiple buggy elements.

Fault localization techniques sometimes assign same suspiciousness score to code elements. Following prior work [39, 40], we use the *worst* ranking for the tied elements. For example, if a buggy element is tied with a correct element in the  $k^{th}$  position of the ranked list, the rank for both elements would be  $k + 1^{th}$ .

## 5.5 Implementation and Tool Supports

For APR, we use PraPR [20], a recent APR technique that fixes bugs at the bytecode level. We choose PraPR because it is one of the most recent APR techniques and has been demonstrated to be able to fix more bugs with a much lower overhead compared to other state-of-the-art techniques. PraPR is set to generate patches for all potentially buggy locations so that the misranked elements can be adjusted by ProFL. Note that, ProFL does not rely on any specific APR technique, since its feedback input (i.e., patch executions) is general and can come from any other APR technique in principle.

We now discuss the collection of all the other information for implementing ProFL and other compared techniques: (i) To collect the coverage information required by SBFL techniques, we use the ASM bytecode manipulation framework [10] to instrument the code on-the-fly via JavaAgent [2]. (ii) To collect the mutation testing information required by MBFL, we use state-of-the-art PIT mutation testing framework [3] (Version 1.3.2) with all its available mutators, following prior MBFL work [39, 40]. Note that we also modify PIT to force it to execute all tests for each mutant and collect detailed mutant impact information (i.e., whether each mutant can impact the detailed test failure message of each test [57]) required by Metallaxis. For PRFL, PRFL<sub>MA</sub>, and DeepFL, we directly used the implementation released by the authors [39, 83]. All experiments are conducted on a Dell workstation with Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, 330GB RAM, and Ubuntu 18.04.1 LTS.

## 5.6 Threats to Validity

**Threats to internal validity** mainly lie in the correctness of implementation of our approach and the compared techniques. To

**Table 5: Overall fault localization results**

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	117	219	259	19.15	24.63
MUSE	89	152	182	47.51	52.81
Metallaxis	84	175	223	17.10	19.54
MCBFL	131	227	267	17.99	23.26
ProFL	161	255	286	9.48	14.37

reduce this threat, we manually reviewed our code and verified that the results of the overlapping fault localization techniques between this work and prior work [40, 82, 83] are consistent. We also directly used the original implementations from prior work [39, 83].

**Threats to construct validity** mainly lie in the rationality of assessment metrics that we chose. To reduce this threat, we chose the metrics that have been recommended by prior studies/surveys [35, 56] and widely used in previous work [39, 40, 67, 82].

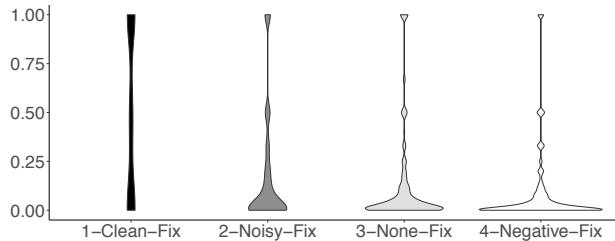
**Threats to external validity** mainly lie in the benchmark suites used in our experiments. To reduce this threat, we chose the widely used Defects4J (V1.2.0) benchmark, which includes hundreds of real bugs collected during real-world software development. To further reduce the threats, compared to previous work, we also conducted the first fault localization evaluation on an extended version of Defects4J, Defects4J (V1.4.0).

## 6 RESULTS

### 6.1 RQ1: Effectiveness of ProFL

To answer this RQ, we first present the overall fault localization results of ProFL and state-of-the-art SBFL and MBFL techniques on Defects4J (V1.2.0) in Table 5. Column “Tech Name” represents the corresponding techniques and the other columns present the results in terms of Top-1, Top-3, Top-5, MFR and MAR. From the table, we observe that ProFL significantly outperforms all the existing techniques in terms of all the five metrics. For example, the Top-1 value of ProFL is 161, 30 more than MCBFL, 44 more than aggregation-based SBFL, 77 more than Metallaxis, and 72 more than MUSE. In addition, MAR and MFR values are also significantly improved (e.g., 47.30% improvement in MFR compared with state-of-the-art MCBFL), indicating a consistent improvement for all buggy elements in the ranked lists. Note that our overall bug ranking results are consistent with prior fault localization work at the method level [40], e.g., state-of-the-art MBFL can outperform SBFL, demonstrating the effectiveness of MBFL. Meanwhile, we observe that SBFL outperforms state-of-the-art MBFL techniques in terms of Top-ranked bugs, which is not consistent with prior work [39]. We find the main reason to be that the prior work did not use suspicious aggregation (which was proposed in parallel with the prior work) for SBFL, demonstrating the effectiveness of suspiciousness aggregation for SBFL.

To further investigate why the simple ProFL approach works, we further analyze each of the four basic ProFL patch categories in a post-hoc way. For each patch category group  $G_i$ , for each bug in the benchmark, we use metric  $Ratio_{ib}$  to represent the ratio of the number of buggy elements (i.e., methods in this work) categorized into group  $G_i$  to the number of all elements categorized into group



**Figure 5:**  $\text{Ratio}_b$  distribution for different patch groups

$\mathcal{G}_i$ . Formally, it can be presented as:

$$\text{Ratio}_b(\mathcal{G}_i) = \frac{|\{e | \mathbb{G}[\mathcal{P}] = \mathcal{G}_i \wedge \mathcal{P} \in \mathbb{P}[e]\} \wedge e \in \mathbb{B}|}{|\{e | \mathbb{G}[\mathcal{P}] = \mathcal{G}_i \wedge \mathcal{P} \in \mathbb{P}[e]\}|} \quad (4)$$

where  $\mathbb{B}$  represents a set of buggy elements.  $\text{Ratio}_b$  ranges from 0 to 1, and a higher value indicates a higher probability for a patch group to contain the actual buggy element(s). We present the distribution of the  $\text{Ratio}_b$  values on all bugs for each of the four different patch groups in the violin plot in Figure 5, where the  $x$  axis presents the four different groups, the  $y$  axis presents the actual  $\text{Ratio}_b$  values, and the width of each plot shows the distribution's density. From the figure we observe that the four different groups have totally different  $\text{Ratio}_b$  distributions. E.g., group CleanFix has rather even distribution, indicating that roughly half of the code elements within this group could be buggy; on the contrary, group NegFix mostly have small  $\text{Ratio}_b$  values, indicating that elements within this group are mostly not buggy. Such group analysis further confirms our hypothesis that different patch categories can be leveraged as the feedback information for powerful fault localization.

**Finding 1:** Simplistic feedback information from program repair can significantly boost existing SBFL-based fault localization techniques, opening a new dimension for fault localization via program repair.

**Table 6: Impacts of finer patch categorization**

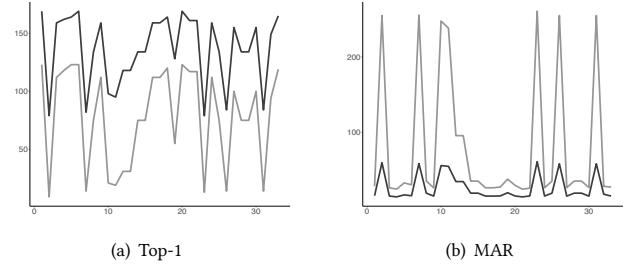
Tech	Top-1	Top-3	Top-5	MFR	MAR
ProFL	161	255	286	9.48	14.37
ProFL <sub>R1</sub>	162	255	286	9.53 ( $p=0.974$ )	14.41 ( $p=0.933$ )
ProFL <sub>R2</sub>	161	252	283	9.56 ( $p=0.904$ )	14.45 ( $p=0.876$ )
ProFL <sub>R3</sub>	161	255	285	9.67 ( $p=0.987$ )	14.62 ( $p=0.899$ )
ProFL <sub>R4</sub>	162	251	285	9.55 ( $p=0.949$ )	14.45 ( $p=0.967$ )

## 6.2 RQ2: Different Experimental Configurations

**6.2.1 RQ2a: Impact of Finer Categorization.** To investigate the four extended rules on the finer categorization presented in Section 4.3, we implemented different ProFL variants based on each rule in Table 3. The experimental results for all the variants are shown in Table 6. In the table, Column “Tech” presents each of the compared variants and the remaining columns present the corresponding metric values computed for each variant. Note that the four variants of ProFL implemented with different rules shown in Table 3 are denoted as ProFL<sub>R1</sub>, ProFL<sub>R2</sub>, ProFL<sub>R3</sub> and ProFL<sub>R4</sub>, respectively.

From the table, we observe that ProFL variants with different extended rules perform similarly with the default setting in all the used metrics. To confirm our observation, we further perform the Wilcoxon signed-rank test [75] (at the significance level of 0.05) to compare each variant against the default setting in terms of both the first and average buggy-method ranking for each bug. The test results are presented in the parentheses in the MFR and MAR columns, and show that there is no significant difference ( $p > 0.05$ ) among the compared variants, indicating that considering the finer-grained grouping does not help much in practice. To explain this observation, we analyze the methods in CleanAllFix, CleanPartFix and find that they have same method sets for over 90% cases, indicating that fixing a subset of failing tests without breaking any passing tests is already challenging.

**Finding 2:** Finer-grained patch grouping has no significant impact on ProFL, further demonstrating the effectiveness of the default grouping.



**Figure 6: Comparison of ProFL and SBFL over all formulae**

**6.2.2 RQ2b: Impact of SBFL Formulae.** Our ProFL approach is general and can be applied to any SBFL formula, therefore, in this RQ, we further study the impact of different SBFL formulae on ProFL effectiveness. The experimental results are shown in Figure 6. In this figure, the  $x$  axis presents all the 34 SBFL formulae considered in this work, the  $y$  axis presents the actual metric values in terms of Top-1 and MAR, while the light and dark lines represent the original SBFL techniques and our ProFL version respectively. We can observe that, for all the studied SBFL formulae, ProFL can consistently improve their effectiveness. For example, the Top-1 improvements range from 41 (for ER1a) to 87 (for GP13), while the MAR improvements range from 36.54% (for Wong) to 77.41% (for GP02). Other metrics follow similar trend, e.g., the improvements in MFR are even larger than MAR, ranging from 49.24% (for SBI) to 80.47% (for GP02). Furthermore, besides the consistent improvement, we also observe that the overall performance of ProFL is quite stable for different SBFL formulae. For example, the MAR value for SBFL has huge variations when using different formulae, while ProFL has stable performance regardless of the formula used, indicating that ProFL can boost ineffective SBFL formulae even more.

**Finding 3:** ProFL can consistently improve all the 34 studied SBFL formulae, e.g., by 49.24% to 80.47% in MFR.

**6.2.3 RQ2c: Impact of Feedback Source.** Since ProFL is general and can even take traditional mutation testing information as feedback

**Table 7: Impacts of using mutation or repair information**

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
MUSE <sub>PIT</sub>	89	152	182	47.51	52.81
MUSE <sub>PraPR</sub>	95	172	207	38.79	43.10
Metallaxis <sub>PIT</sub>	84	175	223	17.10	19.54
Metallaxis <sub>PraPR</sub>	77	170	211	21.42	22.94
MCBFL <sub>PIT</sub>	131	227	267	17.99	23.26
MCBFL <sub>PraPR</sub>	130	228	267	18.03	23.28
ProFL <sub>PIT</sub>	141	238	266	15.24	20.33
ProFL <sub>PraPR</sub>	161	255	286	9.48	14.37

source, we implement a new ProFL variant that directly take mutation information (computed by PIT) as feedback. To distinguish the two ProFL variants, we denote the new variant as ProFL<sub>PIT</sub> and the default one as ProFL<sub>PraPR</sub>. Meanwhile, all the existing MBFL techniques can also take the APR results from PraPR as input (PraPR can be treated as an augmented mutation testing tool with more and advanced mutators), thus we also implemented such variants for traditional MBFL for fair comparison, e.g., the original MUSE is denoted as MUSE<sub>PIT</sub> while the new MUSE variant is denoted as MUSE<sub>PraPR</sub>. Table 7 presents the experimental results for both ProFL and prior mutation-based techniques using different information sources. We have the following observations:

First, ProFL is still the most effective technique compared with other techniques even with the feedback information from mutation testing. For example, ProFL with mutation information localizes 141 bugs within Top-1, while the most effective existing technique (no matter using mutation or repair information) only localizes 131 bugs within Top-1. This observation implies that the ProFL approach of using feedback information (from program-variant execution) to refine SBFL ranking is general in design, and is not coupled tightly with specific source(s) of feedback.

Second, ProFL performs worse when feedback source changes from program repair to mutation testing. For example, the Top-1 decreases from 161 to 141. The reason is that patches within groups CleanFix/NoisyFix can help promote the ranking of buggy methods. However, mutation testing cannot create many such patches. For example, we find that the number of bugs with CleanFix/NoisyFix patches increases by 39.84% when changing from mutation testing to APR. This further indicates that *APR is more suitable than mutation testing for fault localization since it aims to pass more tests while mutation testing was originally proposed to fail more tests*.

Third, for the two existing MBFL techniques, MUSE performs better in program repair compared to mutation testing while Metallaxis is the opposite. We find the reason to be that MUSE simply counts the number of tests changed from passed to failed and vice versa, while Metallaxis leverages the detailed test failure messages to determine mutant impacts. In this way, APR techniques that make more failed tests pass can clearly enhance the results of MUSE, but do not have clear benefits for Metallaxis.

**Finding 4:** ProFL still performs well even with the mutation feedback information, but has effectiveness decrements compared to using program repair, indicating the superiority of program repair over mutation testing for fault localization.

**Table 8: Impacts of using partial matrices**

$\mathbb{M}_p$	Tech Name	Top-1	Top-3	Top-5	MFR	MAR
$\mathbb{M}_p^{(O_1)}$	MUSE <sub>PraPR</sub>	92	148	172	118.56	125.11
	Metallaxis <sub>PraPR</sub>	64	128	167	113.9	126.79
	ProFL	165	260	288	18.18	23.47
$\mathbb{M}_p^{(O_2)}$	MUSE <sub>PraPR</sub>	87	130	152	191.71	206.0
	Metallaxis <sub>PraPR</sub>	32	73	94	163.29	170.61
	ProFL	157	242	281	9.61	14.20
$\mathbb{M}_p^{(O_3)}$	MUSE <sub>PraPR</sub>	89	128	144	169.33	174.09
	Metallaxis <sub>PraPR</sub>	63	127	159	187.19	195.07
	ProFL	155	245	277	19.10	25.19

**6.2.4 RQ2d: Impact of Partial Execution Matrix.** So far, we have studied ProFL using full patch execution matrices. However, in practical program repair, a patch will not be executed against the remaining tests as soon as some test falsifies it for the sake of efficiency. Therefore, we further study new ProFL variants with only partial patch execution matrices. The experimental results for three variants of ProFL using different partial matrices are shown in Table 8. From the table, we have the following observations:

First, surprisingly, ProFL with different partial matrices still perform similarly with our default ProFL using full matrices, while the traditional MBFL techniques perform significantly worse using partial matrices. For example, the MFR and MAR values for existing MBFL all become over 160 when using partial matrices collected following common APR practice (i.e.,  $\mathbb{M}_p^{(O_2)}$ ), while the MFR and MAR values for ProFL have negligible change when using the same partial matrices. We think the reason to be that existing MBFL techniques utilize the detailed number of impacted tests for fault localization and may be too sensitive when switching to partial matrices. Second, ProFL shows consistent effectiveness with partial matrices obtained from different test execution orderings, e.g., even the worst ordering still produces 155 Top-1 bugs. One potential reason that  $\mathbb{M}_p^{(O_3)}$  performs the worst is that if there is any passed tests changed into failing, the original failed tests will no longer be executed, missing the potential opportunities to have CleanFix/NoisyFix patches that can greatly boost fault localization. Luckily, in practice, repair tools always execute the failed tests first (i.e.,  $\mathbb{M}_p^{(O_2)}$ ), further demonstrating that ProFL is practical.

Note that the cost of ProFL consists of two parts: (1) APR time (full/partial matrix collection time), and (2) final fault-localization time based on the APR results. As for the latter cost, ProFL costs less than 2min to compute the suspiciousness scores for each Defects4J (V1.2.0) bug on average (including the APR result parsing time), which is negligible compared to the APR time. Therefore, we next present the cost reduction benefits that partial execution matrices can bring to speed up the ProFL APR time. The experimental results for the HEAD version (i.e., the latest and usually the largest version) of each studied subject are shown in Table 9. In the table, Column “Time<sub>f</sub>” presents the time for executing all tests on each candidate patch while Column “Time<sub>p</sub>” presents the time for terminating test execution on a patch as soon as the patch gets falsified (following the default test execution order of PraPR, i.e.,  $\mathbb{M}_p^{(O_2)}$ , which executes originally failed tests first then passed tests). Similarly, Column “Execution<sub>f</sub>”/“Execution<sub>p</sub>” shows the number

**Table 9: Full and partial matrix collection (with 4 threads)**

Subject	Time <sub>f</sub>	Time <sub>p</sub>	Speedup	Execution <sub>f</sub>	Execution <sub>p</sub>	Speedup
Lang-1	0m38s	0m31s	1.23X	2,282	157	14.54X
Closure-1	2,568m26s	110m33s	23.23X	186,451,071	253,378	735.86X
Mockito-1	452m33s	2m43s	166.58X	4,429,249	8,318	532.49X
Chart-1	32m27s	2m41s	12.09X	796,654	3,769	211.37X
Time-1	149m14s	0m41s	218.39X	677,094	1,147	590.32X
Math-1	68m24s	7m53s	8.63X	244,702	1,162	210.59X
<b>Total</b>	<b>3,271m42s</b>	<b>125m2s</b>	<b>26.17X</b>	<b>192,601,052</b>	<b>267,931</b>	<b>718.85X</b>

of test executions accumulated for all patches for full/partial matrices. From the table, we can observe that partial execution matrix collection can overall achieve 26.17X/718.85X speedup in terms time/test-executions, e.g., even the largest Closure subject only needs less than 2 hours, indicating that ProFL can be scalable to real-world systems (since we have shown that ProFL does not have clear effectiveness drop when using only partial matrices).

**Finding 5:** ProFL keeps its high effectiveness even on partial patch execution matrices, especially with test execution ordering following the program repair practice, demonstrating that its runtime overhead can be reduced by 26.17X without clear effectiveness drop.

**Table 10: Results on Defects4J (V1.4.0)**

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
SBFL	59	102	124	13.81	20.44
MUSE	34	63	73	67.89	74.49
Metallaxis	47	88	115	21.45	28.30
MCBFL	67	112	132	13.20	19.79
ProFL	78	117	131	12.01	17.96

**6.2.5 RQ2e: Impact of Used Benchmarks.** In this RQ, we further compare ProFL and state-of-the-art SBFL/MBFL techniques on additional bugs from Defects4J (V1.4.0), to reduce the threats to external validity. The experimental results are shown in Table 10. From the table, we observe that ProFL still significantly outperforms all other compared techniques. E.g., Top-1 is improved from 59 to 78 compared to the original state-of-the-art SBFL. Such a consistent finding on additional bugs further confirms our findings in RQ1.

**Finding 6:** ProFL still significantly outperforms state-of-the-art SBFL and MBFL on additional bugs.

### 6.3 RQ3: Boosting Learning-Based Localization

We further apply the basic ProFL to boost state-of-the-art unsupervised learning-based (i.e., PRFL and PRFL<sub>MA</sub> [83]) and supervised-learning-based (i.e., DeepFL [39]) fault localization. For unsupervised-learning-based techniques, ProFL is generic and can use any existing fault localization techniques to compute initial suspiciousness (Section 4.2); therefore, we directly apply ProFL on the initial suspiciousness computed by PRFL and PRFL<sub>MA</sub>, denoted as ProFL<sub>PRFL</sub> and ProFL<sub>PRFLMA</sub>, respectively. For supervised-learning-based techniques, ProFL with all the 34 used SBFL formulae can serve as an additional feature dimension; therefore, we augment DeepFL by injecting ProFL features between the original mutation and spectrum feature dimensions (since they are all dynamic features), and denote that as

ProFL<sub>DeepFL</sub>. The experimental results are shown in Table 11. Note that DeepFL results are averaged over 10 runs due to the DNN randomness [39]. First, even the basic ProFL significantly outperforms state-of-the-art unsupervised-learning-based fault localization. E.g., ProFL localizes 161 bugs within Top-1, while the most effective unsupervised PRFL<sub>MA</sub> only localizes 136 bugs within Top-1. Second, ProFL can significantly boost unsupervised-learning-based fault localization. E.g., ProFL<sub>PRFLMA</sub> localizes 185 bugs within Top-1, *the best fault localization results on Defects4J without supervised learning to our knowledge*. Actually, such unsupervised-learning-based fault localization results even significantly outperform many state-of-the-art supervised-learning-based techniques, e.g., TraPT [40], FLUCCS [67], and CombineFL [85] only localize 156, 160, and 168 bugs from the same dataset within Top-1, respectively [39, 85]. Lastly, we can observe that ProFL even boosts state-of-the-art supervised-learning-based technique. E.g., it boosts DeepFL to localize 216.80 bugs within Top-1, *the best fault localization results on Defects4J with supervised learning to our knowledge*. The Wilcoxon signed-rank test [75] with Bonferroni corrections [18] for bug ranking also shows that ProFL significantly boosts all the studied learning-based techniques at 0.05 significance level.

**Finding 7:** ProFL significantly outperforms state-of-the-art unsupervised-learning-based fault localization, and can further boost unsupervised and supervised learning based fault localization, further demonstrating the effectiveness and general applicability of ProFL.

**Table 11: Boosting state-of-the-art learning-based fault localization**

Tech Name	Top-1	Top-3	Top-5	MFR	MAR
PRFL	114	199	243	23.62	27.67
ProFL <sub>PRFL</sub>	179	251	288	10.44	14.83
PRFL <sub>MA</sub>	136	242	269	18.06	22.60
ProFL <sub>PRFLMA</sub>	185	264	295	9.04	13.73
DeepFL	211.00	284.50	310.50	4.97	6.27
ProFL <sub>DeepFL</sub>	216.80	293.60	318.00	4.53	5.88

**Developer patch:**  
Method: public static OrderDTO convertDTOFromAdOrder(AdPlan plan)  
- result.setBizScene(SceneType.APP.toString());  
+ result.setBizScene(plan.getSceneType().getSceneCode());

**Figure 7: Developer patch for Bug-A in industry**

### 6.4 Industry Case Study

ProFL has already been deployed in Alipay [1], a practical online payment system with over 1 billion global users. Before the deployment, the Alipay developers evaluated ProFL on a large number of real bugs and observed that ProFL consistently/largely boosts the Ochiai fault localization that Alipay used (e.g., localizing 2.1X more bugs within Top-1). We now present a case study with one real-world bug that ProFL has recently helped debug within Alipay. The bug is from Project-A (with 100M+ daily users, anonymized according to the company policy), a Spring-style [4] multi-module microservice system (built with Maven) with 197,402 LoC and 418 tests.

State-of-the-art Ochiai and PRFL<sub>MA</sub> localize the buggy method as the 125th and 52nd, respectively. In contrast, within 48min, after exploring 3459 patch executions, the default ProFL (applied on Ochiai with partial PraPR matrices) directly localizes the bug as the 1st, i.e., over 50X improvement in bug ranking. The developers looked into the code and found that although the bug is challenging to automatically fix with the current used repair system PraPR (as shown in Figure 7, this bug requires a multi-edit patch that is not currently supported by PraPR), multiple patches on the actual buggy method was able to mute the bug and make the originally failing tests pass. In this way, ProFL is able to easily point out the actual buggy location.

## 7 CONCLUSION

We have investigated a simple question: *can automated program repair help with fault localization?* To this end, we have designed, ProFL, the first *unified debugging* approach leveraging program repair information as the feedback for powerful fault localization. The results on the widely used Defects4J benchmarks demonstrate that ProFL can significantly outperform state-of-the-art spectrum and mutation based fault localization. Besides, we have demonstrated ProFL’s effectiveness under various settings as well as with an industry case study. Lastly, ProFL even boosts state-of-the-art fault localization via both unsupervised and supervised learning.

## ACKNOWLEDGEMENTS

This work was partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803 and the National Natural Science Foundation of China under Grant Nos. 61872008 and 61861130363. This work was also partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, and Alibaba.

## REFERENCES

- [1] 2019. Alipay. <https://int.alipay.com/>. Accessed Aug-22-2019.
- [2] 2019. JavaAgent. <https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html>
- [3] 2019. Pitest. <http://pitest.org>
- [4] 2019. Spring Framework. <https://spring.io/>. Accessed Jan-10-2020.
- [5] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
- [6] Apache. 2019. Commons Math. <https://commons.apache.org/proper/commons-math/>. Accessed Aug-22-2019.
- [7] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 177–188.
- [8] Antonio Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*. IEEE Computer Society, 85–103.
- [9] Lionel C Briand, Yvan Labiche, and Xuetao Liu. 2007. Using machine learning to support debugging with tarantula. In *ISSRE*. 137–146.
- [10] Eric Bruneton, Romain Lenglet, and Thierry Couppay. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
- [11] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *FSE*. 223–234.
- [12] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based Program Repair Without the Contracts. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (Urbana-Champaign, IL, USA) (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 637–647. <http://dl.acm.org/citation.cfm?id=3155562.3155642>
- [13] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *TSE* 3 (1976), 215–222.
- [14] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for java. In *ECCOP*. 528–550.
- [15] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE ’09)*. IEEE Computer Society, Washington, DC, USA, 550–554. <https://doi.org/10.1109/ASE.2009.15>
- [16] V. DeBroy and W. E. Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *2010 Third International Conference on Software Testing, Verification and Validation*. 65–74. <https://doi.org/10.1109/ICST.2010.66>
- [17] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (1978), 34–41. <https://doi.org/10.1109/C-M.1978.218136>
- [18] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64.
- [19] L. Gazzola, D. Micucci, and L. Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. <https://doi.org/10.1109/TSE.2017.2755013>
- [20] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*. 19–30. <https://doi.org/10.1145/3293882.3330559>
- [21] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Saarbrücken, Germany) (TACAS’11/ETAPS’11). Springer-Verlag, Berlin, Heidelberg, 173–188. <http://dl.acm.org/citation.cfm?id=1987389.1987408>
- [22] Greg4cr. 2019. Defects4J – version 1.4.0. <https://github.com/Greg4cr/defects4j/tree/additional-faults-1.4>.
- [23] Dan Hao, Tao Xie, Lu Zhang, Xiaoyin Wang, Jiasu Sun, and Hong Mei. 2010. Test input reduction for result inspection to facilitate fault localization. *Autom. Softw. Eng.* 17, 1 (2010), 5–31. <https://doi.org/10.1007/s10515-009-0056-x>
- [24] Dan Hao, Lu Zhang, Ying Pan, Hong Mei, and Jiasu Sun. 2008. On similarity-awareness in testing-based fault localization. *Autom. Softw. Eng.* 15, 2 (2008), 207–249. <https://doi.org/10.1007/s10515-008-0025-9>
- [25] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2009. Interactive Fault Localization Using Test Information. *J. Comput. Sci. Technol.* 24, 5 (2009), 962–974. <https://doi.org/10.1007/s11390-009-9270-z>
- [26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. <https://doi.org/10.1109/TSE.2010.62>
- [27] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.
- [28] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11–15, 2019*. IEEE, 502–514. <https://doi.org/10.1109/ASE.2019.00054>
- [29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.
- [30] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. 467–477.
- [31] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) (ISSTA 2014). ACM, New York, NY, USA, 437–440. <https://doi.org/10.1145/2610384.2628055>
- [32] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.
- [33] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
- [34] Edward Kit and Susannah Finzi. 1995. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co.
- [35] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shaping Li. 2016. Practitioners’ expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 165–176.
- [36] Xianglong Kong, Lingming Zhang, W Eric Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *ISSRE*. 194–204.
- [37] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 314–325.

- [38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. <https://doi.org/10.1109/TSE.2011.104>
- [39] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 169–180. <https://doi.org/10.1145/3293882.3330574>
- [40] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 92.
- [41] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 – June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 82–92. <https://doi.org/10.1145/3180155.3180242>
- [42] Ben Lliblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. *PLDI* (2005), 15–26.
- [43] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jin Song Dong. 2017. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017*, 393–403. <https://doi.org/10.1109/ICSE.2017.43>
- [44] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 – September 4, 2015*, 166–178. <https://doi.org/10.1145/2786805.2786811>
- [45] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 – 22, 2016*, 298–312. <https://doi.org/10.1145/2837614.2837617>
- [46] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 43–54.
- [47] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, 269–278.
- [48] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (01 Aug 2017), 1936–1964. <https://doi.org/10.1007/s10664-016-9470-4>
- [49] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016*, 691–701. <https://doi.org/10.1145/2884781.2884807>
- [50] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. <https://doi.org/10.1145/3105906>
- [51] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, IEEE, 153–162.
- [52] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.
- [53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013*, 772–781. <https://doi.org/10.1109/ICSE.2013.6606623>
- [54] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate "unknown" faults. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, IEEE, 691–700.
- [55] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5–7 (2015), 605–628.
- [56] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. ACM, 199–209.
- [57] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Derik Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, 609–620.
- [58] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (2014), 427–449. <https://doi.org/10.1109/TSE.2014.2312918>
- [59] William E Perry. 2007. *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. John Wiley & Sons.
- [60] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyi Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (Hyderabad, India) (ICSE 2014)*. ACM, New York, NY, USA, 254–265. <https://doi.org/10.1145/2568225.2568254>
- [61] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, 24–36.
- [62] Shounak Roychowdhury and Sarfraz Khurshid. 2011. A novel framework for locating software faults using latent divergences. In *ECML*, 49–64.
- [63] Shounak Roychowdhury and Sarfraz Khurshid. 2011. Software fault localization using feature selection. In *International Workshop on Machine Learning Technologies in Software Engineering*, 11–18.
- [64] Shounak Roychowdhury and Sarfraz Khurshid. 2012. A family of generalized entropies and its application to software fault localization. In *International Conference Intelligent Systems*, 368–373.
- [65] Ripon K Saha, Yingjin Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 648–659.
- [66] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).
- [67] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 273–283.
- [68] Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project 7007*, 011 (2002).
- [69] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An investigation into the use of mutation analysis for automated program repair. In *International Symposium on Search Based Software Engineering*. Springer, 99–114.
- [70] Tricentis. 2019. "Tricentis Report". <https://www.tricentis.com>. "accessed 10-jan-2020".
- [71] Westley Weimer. 2006. Patches As Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (Portland, Oregon, USA) (GPCE '06)*. ACM, New York, NY, USA, 181–190. <https://doi.org/10.1145/1173706.1173734>
- [72] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*, 356–366.
- [73] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, 1–11.
- [74] Wikipedia contributors. 2019. Software bug – Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/wiki/Software\\_bug](https://en.wikipedia.org/wiki/Software_bug) [accessed 10-jan-2020].
- [75] Wikipedia contributors. 2019. Wilcoxon signed-rank test – Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/wiki/Wilcoxon\\_signed-rank\\_test](https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test) [accessed 10-jan-2020].
- [76] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.* 42, 8 (Aug. 2016), 707–740. <https://doi.org/10.1109/TSE.2016.2521368>
- [77] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *ICSE*, 808–819.
- [78] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. <https://doi.org/10.1109/TSE.2016.2560811>
- [79] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 191–200.
- [80] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *FSE*, 52–63.
- [81] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, 765–784.
- [82] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 261–272.
- [83] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank. *IEEE Transactions on Software Engineering* (2019).
- [84] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. ACM, 272–281.
- [85] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* (2019). <https://doi.org/10.1109/TSE.2019.2892102>

# Automated Repair of Feature Interaction Failures in Automated Driving Systems

Raja Ben Abdessalem  
University of Luxembourg  
Luxembourg  
raja.benabdessalem@uni.lu

Annibale Panichella  
Delft University of Technology  
Netherlands  
University of Luxembourg  
Luxembourg  
a.panichella@tudelft.nl

Shiva Nejati  
University of Ottawa  
Canada  
University of Luxembourg  
Luxembourg  
snejati@uottawa.ca

Lionel C. Briand  
University of Luxembourg  
Luxembourg  
University of Ottawa  
Canada  
lionel.briand@uni.lu

Thomas Stifter  
IEE S.A.  
Luxembourg  
thomas.stifter@iee.lu

## ABSTRACT

In the past years, several automated repair strategies have been proposed to fix bugs in individual software programs without any human intervention. There has been, however, little work on how automated repair techniques can resolve failures that arise at the system-level and are caused by undesired interactions among different system components or functions. Feature interaction failures are common in complex systems such as autonomous cars that are typically built as a composition of independent features (i.e., units of functionality). In this paper, we propose a repair technique to automatically resolve undesired feature interaction failures in automated driving systems (ADS) that lead to the violation of system safety requirements. Our repair strategy achieves its goal by (1) localizing faults spanning several lines of code, (2) simultaneously resolving multiple interaction failures caused by independent faults, (3) scaling repair strategies from the unit-level to the system-level, and (4) resolving failures based on their order of severity. We have evaluated our approach using two industrial ADS containing four features. Our results show that our repair strategy resolves the undesired interaction failures in these two systems in less than 16h and outperforms existing automated repair techniques.

## CCS CONCEPTS

- Software and its engineering → Software verification and validation; Search-based software engineering.

## KEYWORDS

Search-based Software Testing, Automated Driving Systems, Automated Software Repair, Feature Interaction Problem

## ACM Reference Format:

Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2020. Automated Repair of Feature Interaction Failures in Automated Driving Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397386>

## 1 INTRODUCTION

Software for automated driving systems (ADS) is often composed of several units of functionality, known as ADS features, that automate specific driving tasks (e.g., automated emergency braking, automated traffic sign recognition, and automated cruise control). Each feature executes continuously over time and receives data from the perception layer components. The latter often use artificial intelligence (e.g., deep neural networks [29]) to extract –based on sensor data—environment information, such as the estimated position of the (ego) car, the road structure and lanes, and the position and speed of pedestrians. Using this information, ADS features independently determine the car maneuver for the next time step.

Some maneuvers issued by different features may be conflicting. For example, automated cruise control may order the (ego) car to accelerate to follow the speed of the leading car while, at the same time, the automated traffic sign recognition feature orders the (ego) car to stop since the car is approaching an intersection with a traffic light that is about to turn red. To resolve conflicting maneuvers, *integration rules* are applied. Integration rules are conditional statements that determine under what conditions which feature outputs should be prioritized. They are typically defined manually by engineers based on their domain knowledge and are expected to ensure safe and desired car maneuvering. In the above example, integration rules should prioritize the braking maneuver of automated traffic sign recognition when the car can stop safely

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA  
© 2020 Association for Computing Machinery.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397386>

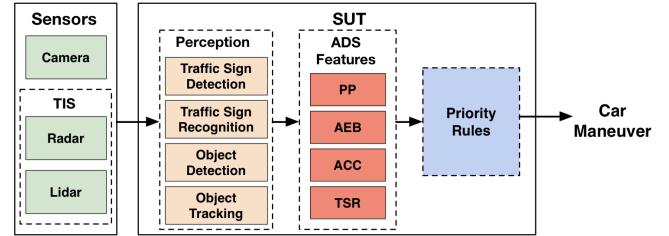
before the intersection, and otherwise should prioritize the maneuver of the automated cruise control. Integration rules tend to become complex as they have to address a large number of different environment and traffic situations and predict how potential conflicts should be resolved in each situation. As a result, the rules may be faulty or incomplete, and there is a need to assist engineers to *repair* integration rules so that, at each time step and for each situation, they prioritize the maneuver that can ensure ADS safety.

Recently, many approaches have been proposed in the literature to repair faults in software code automatically [15, 22, 37, 44]. The inputs to these techniques are a faulty program and a test suite that contains some passing test cases capturing the desired program behavior and some failing test cases revealing a fault that needs to be fixed. Fault-repair techniques iteratively identify faulty statements in the code (fault localization [20, 45]), automatically modify the faulty statements (patch generation) and check if the patched code passes all the input test cases (patch validation). To effectively repair ADS integration faults, we need a technique, that in addition to supporting the three above steps, ensures the following objectives:

**O1. The repair technique should localize faults spanning multiple lines of code.** Most program repair techniques localize faults using spectrum-based techniques. These techniques rely on the assumption that there is a single faulty statement in the program [45, 46], and as shown by Pearson et al. [36], they may not identify all defective lines in multi-statements faults. As we describe in Section 3, faults in integration rules are often multi-location as they may span multiple lines of code, for example, when the rules are applied in a wrong order.

**O2. The repair technique should be able to fix multiple independent faults in the code.** Most program repair techniques assume that all failing test cases in the given (input) test suite fail due to the same fault in the code. Hence, they generate a single patch for a single fault at a time [28, 38, 43]. The generated patches may replace a single program statement [28, 38, 43] or they may be built based on predefined templates, prescribing hard-coded change patterns to modify programs in multiple locations [30, 31]. For ADS integration rules, however, the assumption that there is a single fault in the code does not hold since test cases may fail due to the presence of multiple faults located in different parts of the code. As we describe in Section 3, to fix all the failing test cases in a given test suite, we may have to fix more than one fault in the code by, for example, reordering some rules, and in addition, changing some clauses in preconditions of some other rules.

**O3. The repair technique should scale to fixing faults at system-level where the software under test is executed using a feedback-loop simulator.** Testing ADS software requires a simulation-based test platform that can execute the ADS software for different road structures, weather conditions and traffic situations. Testing ADS software using simulators presents two challenges: First, each test case is a scenario that executes over a time period and runs the ADS software through a feedback-loop simulation. Hence, a single test case runs the ADS software at every simulation time step and generates an array of outputs over time instead of a single output. As a result, dependencies in ADS software are both structural (like typical code) and temporal—a *change in the output at time t impacts the input of the system at time t + 1 through the feedback-loop simulation which may impact the*



**Figure 1: An overview of a self-driving system, *AutoDrive*, consisting of four ADS features.**

*behaviour of the system in all subsequent time steps after t.* Second, test cases for ADS software are computationally expensive since each test case has to execute for a time duration and has to execute a complex simulator and the ADS software at each time step. This poses limitations on the sizes of test suites used for repair and the number of intermediary patches that we can generate.

**O4. The repair technique should resolve failures in their order of severity.** When testing generic software, test cases are either passing or failing, and there is no factor distinguishing failing test cases from one another. For ADS, however, some failures are more critical than others. For example, a test case violating the speed limit by 10km/h is more critical than the one violating the speed limit by 5km/h. Similarly, a test scenario in which a car hits a pedestrian with a speed lower than 10km/h is less critical than the ones where collisions occur at a higher speed. Our repair strategy should, therefore, assess failing test cases in a more nuanced way and prioritizes repairing the more critical failing test cases over the less critical ones.

**Contributions.** We propose an approach to Automated Repair of IntEgration ruLes (ARIEL) for ADS. Our repair strategy achieves the four objectives O1 to O4 described above by relying on a many-objective, single-state search algorithm that uses an archive to keep track of partially repaired solutions. We have applied ARIEL to two industry ADS case studies from our partner company [name redacted]. Our results show that ARIEL is able to repair integration faults in our two case studies in five and eleven hours on average. Further, baseline repair strategies based on Genetic Programming [26] and Random Search either never succeed to do so or repair faults with a maximum probability of 60% across different runs, i.e., in 12 out of 20 runs. Finally, we report on the feedback we received from engineers regarding the practical usefulness of ARIEL. The feedback indicates that ARIEL is able to generate patches to integration faults that are understandable by engineers and could not have been developed by them without any automation and solely based on their expertise and domain knowledge.

**Structure.** Section 2 provides motivation and background. Section 3 presents our approach. Section 4 describes our evaluation. Section 6 discusses the related work. Section 7 concludes the paper.

## 2 MOTIVATION AND BACKGROUND

We first motivate our work using a self-driving system case study, *AutoDrive*, from our industry partner (company A). We then formalize the concepts required in our approach.

## 2.1 Motivating Case Study

Figure 1 shows an overview of *AutoDrive* that consists of four ADS features: Autonomous Cruise Control (ACC), Traffic Sign Recognition (TSR), Pedestrian Protection (PP), and Automated Emergency Braking (AEB). ACC automatically adjusts the car speed and direction to maintain a safe distance from the leading car. TSR detects traffic signs and applies appropriate braking, acceleration, or steering commands to follow the traffic rules. PP detects pedestrians in front of a car with whom there is a risk of collision and applies a braking command if needed. AEB prevents accidents with objects other than pedestrians. Since these features generate braking, acceleration and steering commands to control the vehicle’s actuators, they may send conflicting commands to the actuators at the same time. For example, ACC orders the car to accelerate to follow the speed of the leading car, while, at the same time, a pedestrian starts crossing the road. Hence, PP starts sending braking commands to avoid hitting the pedestrian.

Automated driving systems use integration rules to resolve conflicts among multiple maneuvers issued by different ADS features. The rules determine what feature output should be prioritized at each time step based on the environment factors and the current state of the system. Figure 2 shows a *small subset* of integration rules for AutoDrive. The rules are simplified for the sake of illustration. Each rule activates a single ADS feature (i.e., prioritizes the output of a feature over the others) when its *precondition*, i.e., the conjunctive clauses on the left-hand side of each rule, holds. All the rules are checked at every time step  $t$ , and the variables in the rules are indexed by time to indicate their values at a specific time step  $t$ . The rules in Figure 2 include the following environment variables: *pedestrianDetected* determines if a pedestrian with whom the car may risk a collision is detected; *ttc* or time to collision is a well-known metric in self-driving systems measuring the time required for a vehicle to hit an object if both continue with the same speed [40]; *dist(p, c)* denotes the distance between the car  $c$  and the pedestrian  $p$ ; *dist(c, sign)* denotes the distance between the car  $c$  and a traffic sign  $sign$ ; *speed* is the speed of the ego car, i.e., the car with the self-driving software; *speedLeadingCar* is the speed of the car in front of the ego car; *objectDetected* determines if an object which cannot be classified as a pedestrian is detected in front of the car, and *stopSignDetected* that determines if a stop sign is detected. For example, **rule1** states that feature PP is prioritized at time  $t$  if a pedestrian in front of the car is detected, the car and pedestrian are close ( $\text{dist}(p, c)(t) < \text{dist}_{th}$ ) and a collision is likely ( $\text{ttc}(t) < \text{ttc}_{th}$ ). Note that  $\text{dist}_{th}$ ,  $\text{dist}_{cs}$  and  $\text{ttc}_{th}$  are threshold values. Dually, **rule2** states that if there is a risk of collision with another object different from a pedestrian ( $\text{ttc}(t) < \text{ttc}_{th} \wedge \text{objectDetected}(t) \wedge \neg \text{pedestrianDetected}(t) \wedge \text{objectDetected}(t)$ ), then AEB should be applied. Note that the braking command issued by AEB is less intense than that issued by PP.

Integration rules are likely to be erroneous, leading to system safety requirements violations. In particular, we identify two general ways where the integration rules may be wrong: (1) The preconditions of the rules may be wrong. Specifically, there might be missing clauses in the preconditions, or the thresholds used in some clauses may not be accurate or there might be errors in mathematical or relational operators used in the preconditions (i.e., using  $<$

At every time step $t$ apply:	
rule1:	$(\text{ttc}(t) < \text{ttc}_{th}) \wedge (\text{dist}(p, c)(t) < \text{dist}_{th}) \wedge \text{pedestrianDetected}(t) \implies \text{PP.active}(t)$
rule2:	$(\text{ttc}(t) < \text{ttc}_{th}) \wedge \neg \text{pedestrianDetected}(t) \wedge \text{objectDetected}(t) \implies \text{AEB.active}(t)$
rule3:	$\text{speed}(t) < \text{speedLeadingCar}(t) \implies \text{ACC.active}(t)$
rule4:	$(\text{speed}(t) > \text{speed-limit}) \wedge (\text{dist}(c, sign)(t) < \text{dist}_{cs}) \implies \text{TSR.active}(t)$
rule5:	$\text{stopSignDetected}(t) \implies \text{TSR.active}(t)$

**Figure 2: Ordered integration rules to resolve conflicts between different ADS features.**

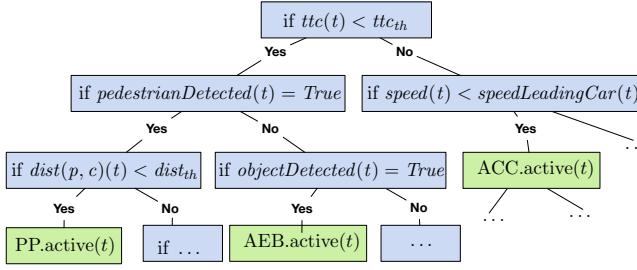
instead of  $>$  or  $+$  instead of  $-$ ). For example, if the threshold  $\text{dist}_{th}$  in **rule1** is too small, we may activate PP only when the car is too close to the pedestrian with little time left to avoid an accident. (2) The integration rules may be applied in a wrong order. Applying the rules in different orders leads to different system behaviors, some of which may violate safety requirements. For example, **rule3** and **rule4** can be checked in two different orders. However, applying **rule3** first may lead to a scenario where the car keeps increasing its speed since ACC orders to follow a fast driving leading car, and it violates the speed-limit as **rule4**, being at a lower priority than **rule3**, is never checked and cannot activate TSR. To avoid this situation, **rule4** should be prioritized over **rule3**.

## 2.2 Context Formalization

We define an ADS as a tuple  $(f_1, \dots, f_n, \Pi)$ , where  $f_1, \dots, f_n$  are features and  $\Pi$  is a decision tree diagram capturing the system integration rules. For example, Figure 3 shows a fragment of a decision tree diagram related to the rules in Figure 2. Each internal node of  $\Pi$  corresponds to a clause of a rule precondition and the tree leaves are labelled by features. Specifically, each path  $\pi$  of  $\Pi$  corresponds to one integration rule *precondition*  $\implies f$  such that the leaf of  $\pi$  is labelled by  $f$  and the conjunction of clauses appearing on the non-leaf nodes of  $\pi$  are equal to *precondition*. We denote by  $f(\pi_i)$  the label of the leaf of  $\pi_i$ , i.e., the feature activated when  $\pi_i$  is executed. A set  $\Pi$  of integration rules are implemented using a set  $S$  of program statements. Each statement  $s \in S$  corresponds to a node of some path  $\pi \in \Pi$ . We use the notation  $s \in \pi$  when statement  $s$  corresponds to some node on  $\pi$ . The conditional statements correspond to non-leaf nodes of  $\Pi$  and have this general format:  $\langle \text{if } op1 \text{ operator threshold} \rangle$  where  $op1$  is a variable, operator is a relational operator and threshold is boolean or numeric constant (see Figure 3)

Let  $SR = \{r_1, \dots, r_k\}$  be a set of safety requirements. Due to the modular structure of ADS, each requirement is related to one feature, which is responsible for the satisfaction of that requirement. For example, Table 1 shows the set of requirements for *AutoDrive* and the feature responsible for satisfying each requirement.

We denote by  $TS = \{tc_1, \dots, tc_l\}$  a set of test cases for an ADS. Given that ADS testing is performed via simulators, each test  $tc_i$  specifies initial conditions and parameters for the simulator to mimic a specific self-driving scenario. To run a test case, the simulator that is initialized using a test case is executed for a time duration  $T$  set by the user. The simulator computes the state of the system at regular and fine-grained steps. Let  $\delta$  be the simulation



**Figure 3: A fragment of the decision tree diagram related to the integration rules in Figure 2.**

**Table 1: Safety requirements for AutoDrive.**

Features	Requirements
PP	The PP system shall avoid collision with pedestrians by initiating emergency braking in case of impending collision with pedestrians.
TSR	The TSR system shall stop the vehicle at the stop sign by initiating a full braking when a stop sign is detected
AEB	The AEB system shall avoid collision with vehicles by initiating emergency braking in case of impending collision with vehicles.
ACC	The ACC system shall respect the safety distance by keeping the vehicle within a safe distance behind the preceding vehicle.
TSR	The TSR system shall respect the speed limit by keeping the vehicle's speed at or below the road's speed limit.

time step size. The simulator computes the test results as vectors of size  $q$  where  $T = q \cdot \delta$ . Each element  $i$  s.t.  $0 \leq i \leq q$  of these vectors indicates the test output at the  $i$ th time step.

At each time step  $u \in \{0, \dots, q\}$ , each test case  $tc$  executes the integration rules  $\Pi$  that select a single feature to be applied at that step. We write  $tc(u)$  to refer to the output of  $tc$  at step  $u$ . We define an oracle function  $O$  to assess test outputs with respect to the system safety requirements  $SR$ . Specifically,  $O(tc_i(u), r_j)$  assess the output of  $tc_i$  at time step  $u$  with respect to requirement  $r_j$ . We define  $O$  as a quantitative function that takes a value between  $[-1, 1]$  such that a negative or zero value indicates that the test case fails and a positive value implies that the test case passes. The quantitative oracle value allows us to distinguish between different degrees of satisfaction and failure (objective **O4**). When  $O$  is positive, a closer value to 0 indicates that we are close to violating, although not yet violating, some safety requirement, while a value close to 1 shows that the system is far from violating its requirements. Dually, when  $O$  is negative, an oracle value close to 0 shows a less severe failure than a value close to -1. We define the oracle function  $O$  for every requirement  $r_j \in SR$  separately such that  $O$  ensure the above properties. For example, for the safety requirement related to PP (denoted  $r_{PP}$ ), we define  $O(tc_i(u), r_{PP})$  for every test case  $tc_i$  and every time step  $u$  as a normalized value of  $(dist(p, c)(u \cdot \delta) - dist_{fail})$  where  $dist(p, c)$  is the same variables used in rule1 in Figure 2, and  $dist_{fail}$  is a threshold value indicating the smallest distance allowed between a car and a pedestrian below which  $r_{PP}$  is violated.

In our work, we select the test suite  $TS$  such that it can cover all the paths in  $\Pi$ . That is, for each path  $\pi_i \in \Pi$  there is some test case  $tc_j \in TS$  and some time step  $u$  such that  $tc_j(u)$  executes  $\pi_i$ , i.e., test case  $tc_j$  executes the path  $\pi_i$  at step  $u$ . Given a test case  $tc_j \in TS$ , we say  $tc_j$  fails, if it violates some safety requirement at some time step

$u$ , i.e., if there is a requirement  $r_j$  and some time step  $u$  such that  $O(tc_j(u), r_j) \leq 0$ . We refer to the time step  $u$  at which  $tc_j$  fails as *the time of failure* for  $tc_j$ . Otherwise, we say  $tc_j$  passes. We denote by  $TS_f$  and  $TS_p$  the set of failing and passing test cases, respectively. In ADS, a test case may fail due to a fault either in a feature or in integration rules. In order to ensure that a failure revealed by a test case  $tc$  is due to a fault in integration rules, we can remove integration rules and test features individually using  $tc$ . If the same failure is not observed, we can conclude that integration rules are faulty. In addition, we can combine our oracle function  $O$  with heuristics provided by the feature interaction analysis research [9] to distinguish failures caused by integration rules from those caused by errors inside features.

### 3 APPROACH

In this section, we present our approach, Automated Repair of Integration rules (ARIEL), to locate and repair faults in the integration rules of ADS. The inputs to ARIEL are (1) a faulty ADS  $(f_1, \dots, f_n, \Pi)$  and (2) a test suite  $TS$  verifying the safety requirements  $SR = \{r_1, \dots, r_k\}$ . The output is a repaired ADS, i.e.,  $(f_1, \dots, f_n, \Pi^*)$ , where  $\Pi^*$  are the repaired integration rules. Below, we present different steps of ARIEL in detail.

#### 3.1 Fault Localization

Feature interaction failures may be caused by multiple independent faults in integration rules, and some of these faults may span over multiple statements of the code of integration rules. As shown by a prior study [36], state-of-the-art fault localization methods often fail to pinpoint all faulty lines (i.e., those to mutate in program repair). Further, recall from Section 1 that to test ADS, we have to execute the system within a continuous loop over time. Hence, each ADS test case covers multiple paths of  $\Pi$  over the simulation time duration, i.e., one path  $\pi \in \Pi$  is covered at each time step  $u$ . Hence, failing and non-failing tests cover large, overlapping subsets of paths in  $\Pi$  albeit with different data values. In this case, applying existing fault localization formulae (e.g., Tarantula and Ochiai) results in having most statements in  $\Pi$  with the same suspiciousness score, leaving the repair process with little information as to exactly where in  $\Pi$  mutation/patches should be applied.

Based on these observations, we modify the Tarantula formula by (1) focusing on the path  $\pi \in \Pi$  covered at the time of failure, and (2) considering the “severity” of failures. There are alternative fault localization formulae that we could use. In this paper, we focus on Tarantula since a previous study [36] showed that the differences among alternative methods (e.g., Tarantula and Ochiai) are negligible when applied to real faults. Below, we describe how we modified Tarantula to rank the faulty paths in  $\Pi$ .

**3.1.1 Localizing the Faulty Path.** Given a failure revealed by a test case  $tc_i \in TS$ , the goal is to determine the path executed by  $tc_i$  that is more likely to have caused the failure. At each time step  $u$ , one path  $\pi_i$  is executed, and only one feature (i.e.,  $f(\pi_i)$ ) is selected. A failure that arises at time step  $u'$  is likely to be due to faults in the integration rules of the path  $\pi_i$  that is executed at  $u'$ . As discussed in Section 2, we distinguish two types of faults: (1) *wrong conditions*, that are faults in the rules’ preconditions, e.g., the braking command is activated when the distance between the car

and the pedestrian is too small, and (2) *wrong ordering of rules*, that are faults due to incorrect ordering of the rules activating features. For the purpose of fault localization, we only consider the path  $\pi_i \in \Pi$  executed at the time of the failure ( $u'$ ) as the execution trace for fault localization, and ignore all other paths covered by a failing test case before and after the failure. In contrast, in traditional fault localization, all statements (rules in our case) covered by the test cases should be considered.

**3.1.2 The Severity of the Failure.** In Tarantula (as well as in state-of-the-art fault localization techniques), all failing tests have the same weight (i.e., one) in the formula for computing statement suspiciousness. However, in case of multiple faults, this strategy does not help prioritize the repair of the (likely) faulty statements. Focusing on faulty statements related to the most severe failures can lead to patches with the largest potential gains in fitness. As explained in Section 2, we can measure how severely a safety requirement of the ADS is violated. The severity corresponds to the output  $O(tc_i(u), r_j)$  of the test case  $tc_i$  at the time stamp  $u$ . For example, let us consider the safety requirement: “*the minimum distance between the ego car and the pedestrian must be larger than a certain threshold  $dist_{fail}$* ”. A failure happens when the distance between the car and the pedestrian falls below the threshold  $dist_{fail}$ . This corresponds to having a test output  $O(tc_i(u), r_{PP})$  within the interval  $[-1; 0]$  for the test case  $tc_i$  and requirement  $r_{PP}$ . The lower the output value, the larger the severity of the violations for  $r_{PP}$ .

**3.1.3 Our Fault Localization Formula.** Based on the observations above, we define the suspiciousness of each statement  $s$  by considering both failure severity and faulty paths executed by failing test cases when the tests fail. Note that for each failing test case  $tc$ , there is a unique time step  $u$  at which  $tc$  fails (i.e., when we have  $O(tc(u), r) \leq 0$  for some requirement  $r$ ). The execution of  $tc$  stops as soon as it fails. In fact, the time of failure for a failing test case  $tc$  is the last time step of the execution of  $tc$ . Given a failing test  $tc \in TS_f$ , we write  $cov(tc, s) \in \{0, 1\}$  to denote whether, or not,  $tc$  covers a statement  $s$  at its last step (when it fails). Further, we denote by  $w_{tc}$  the weight (severity) of the failure of  $tc$ . We then compute the suspiciousness of each statement  $s$  as follows:

$$Susp(s) = \frac{\sum_{tc \in TS_f} [w_{tc} \cdot cov(tc, s)]}{\sum_{tc \in TS_f} w_{tc}} \quad (1)$$

$$\frac{\text{passed}(s)}{\text{total\_passed}} + \frac{\text{failed}(s)}{\text{total\_failed}}$$

where  $\text{passed}(s)$  is the number of passed test cases that have executed  $s$  at some time step;  $\text{failed}(s)$  is the number of failed test cases that have executed  $s$  at some time step; and  $\text{total\_passed}$  and  $\text{total\_failed}$  denote the total numbers of failing and passing test cases, respectively. Note that Equation 1 is equivalent to the standard Tarantula formula if we let the weight (severity) for failing test cases be equal to one (i.e., if  $w_{tc} = 1$  for every  $tc \in TS_f$ ):

$$Susp(s) = \frac{\frac{\text{failed}(s)}{\text{total\_failed}}}{\frac{\text{passed}(s)}{\text{total\_passed}} + \frac{\text{failed}(s)}{\text{total\_failed}}} \quad (2)$$

For each test case  $tc$  that fails at time step  $u$  and violates some requirement  $r$ , we define  $w_{tc} = |O(tc(u), r)|$ . That is,  $w_{tc}$  is the degree of violation caused by  $tc$  at the time step  $u$  when it fails. Hence, we

assign larger weights (larger suspiciousness) to statements covered by test cases that lead to more severe failures (addressing O4 in Section 1). Note that each test case violates at most one requirement since the simulation stop as soon as a requirement is violated.

### 3.2 Program Repair

Traditional evolutionary tools, such as GenProg, use population-based algorithms (e.g., Genetic programming), which evolves a pool of candidate patches iteratively. Each candidate patch is evaluated against the entire test suite to check whether changes lead to more or fewer failing tests. Hence, the overall cost of one single iteration/generation is  $N \times \sum_{tc \in TS} cost(tc)$ , where  $cost(tc)$  is the cost of the test  $tc$  and  $N$  is the population size.

Population-based algorithms assume that the cost of evaluating individual patches is not large, and hence, test results can be collected within a few seconds and used to give feedback (i.e., compute the fitness function) to the search. However, as discussed in O3 in Section 1, this assumption does not hold in our context, and it takes in the order of some minutes to run a single simulation-based test case. Hence, evaluating a pool of patches in each iteration/generation becomes too expensive (in the order of hours).

Based on the observations above, we opted for the (1+1) Evolutionary Algorithm (EA) with an archive. (1+1) EA selects only one parent patch and generates only one offspring patch in each generation. Generating and assessing only one patch at a time allows to reduce the cost of each iteration, thus addressing O3 in Section 1. Our program repair approach, ARIEL, includes (i) (1+1) EA, (ii) our fault localization formula (Equation 1), and (iii) the archiving strategy. Algorithm 1 provides the pseudo-code of ARIEL.

ARIEL receives as input the test suite  $TS$  and the faulty self-driving system  $(f_1, \dots, f_n, \Pi)$ , where  $\Pi$  denotes the faulty integration rules. The output are a set of repaired integration rules (denoted by  $\Pi^*$ ) that pass all test cases in  $TS$ . The algorithm starts by initializing the *archive* with the faulty rules  $\Pi$  (line 2) and computing the objective scores, which we describe in Section 3.2.3 (line 3). Then, the archive is evolved iteratively within the loop in lines 4-8. In each iteration, ARIEL selects one patch  $\Pi_p \in \text{archive}$  randomly (line 5) and creates one offspring  $(\Pi_o)$  in line 6 (routine GENERATE-PATCH) by (1) applying fault localization (Equation 1) and (2) mutating the rules in  $\Pi_p$ . The routine GENERATE-PATCH is presented in subsection 3.2.1.

Then, the offspring  $\Pi_o$  is evaluated (line 7) by running the test suite  $TS$ , extracting the remaining failures, and computing their corresponding objective scores ( $\Omega$ ). Note that the severities of the failures are our search objectives and are further discussed in Section 3.2.3. The offspring  $\Pi_o$  is added to the *archive* (line 8 of Algorithm 1) if it decreases the severity of the failures compared to the patches currently stored in the *archive*. The *archive* and its updating routine are described in details in subsection 3.2.4. The search stops when the termination criteria are met (see Section 3.2.5).

**3.2.1 Generating a Patch.** ARIEL generates patches using the routine in Algorithm 2. First, it applies our fault localization formula (routine FAULT-LOCALIZATION in line 2) to determine the suspiciousness of the statements in  $\Pi$  and based on the test results.

To select a statement  $s \in \Pi$  among the most suspicious ones, we use the Roulette Wheel Selection (RWS) [16], which assigns

**Algorithm 1:** ARIEL

---

**Input:**  
 $(f_1, \dots, f_n, \Pi)$ : Faulty self-driving system  
 $TS$ : Test suite  
**Result:**  $\Pi^*$ : repaired integration rules satisfying all  $tc \in TS$

```

1 begin
2   Archive ←  $\Pi$ 
3    $\Omega \leftarrow \text{RUN-EVALUATE}(\Pi, TS)$ 
4   while  $\text{not}(|\text{Archive}|=1 \& \text{Archive satisfies all } tc \in TS)$  do
5      $\Pi_p \leftarrow \text{SELECT-A-PARENT}(\text{Archive})$  // Random selection
6      $\Pi_o \leftarrow \text{GENERATE-PATCH}(\Pi_p, TS, \Omega)$ 
7      $\Omega \leftarrow \text{RUN-EVALUATE}(\Pi_o, TS)$ 
8      $\text{Archive} \leftarrow \text{UPDATE-ARCHIVE}(\text{Archive}, \Pi_o, \Omega)$ 
9   return Archive

```

---

**Algorithm 2:** GENERATE-PATCH

---

**Input:**  
 $\Pi$ : A faulty rule-set  
 $TS$ : Test suite  
 $\Omega$ : Severity of failures (Objectives Scores)  
**Result:**  $\Pi_o$ : A mutated  $\Pi$

```

1 begin
2    $\{\pi, s\} \leftarrow \text{FAULT-LOCALIZATION}(\Pi, TS, \Omega)$  // Section 3.1
3    $counter \leftarrow 0$ 
4    $p = \text{RANDOM-NUMBER}(0,1)$  //  $p \in [0, 1]$ 
5    $\Pi_o \leftarrow \Pi$ 
6   while  $p \leq 0.5^{counter}$  do
7      $\Pi_o \leftarrow \text{APPLY-MUTATION}(\Pi_o, \pi, s)$ 
8      $counter \leftarrow counter + 1$ 
9      $p = \text{RANDOM-NUMBER}(0,1)$  //  $p \in [0, 1]$ 
10  return  $\Pi_o$ 

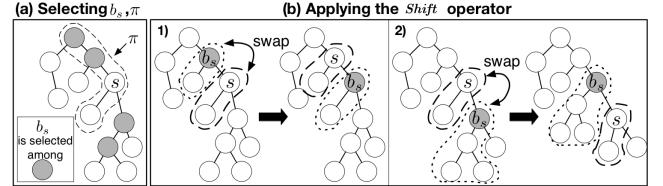
```

---

each statement a probability of being selected for mutation. The probabilities are based on the suspiciousness of each statement (Equation 1). More specifically, the probability of a statement  $s \in \Pi$  is  $\text{prob}(s) = \frac{\text{Susp}(s)}{\sum_{s_i \in \Pi} \text{Susp}(s_i)}$ . The higher the suspiciousness of a statement  $s$ , the higher its probability of being selected for mutation. The routine FAULT-LOCALIZATION returns (1) the statement  $s$  to be mutated, which is randomly selected based on RWS, and (2) a path  $\pi \in \Pi$  such that  $s \in \pi$  (i.e.,  $s$  is on path  $\pi$ ) and  $\pi$  is executed by some failing test case at its last time step (i.e.,  $\pi$  is executed at the time of failure of some test case). If several paths satisfy the latter condition, FAULT-LOCALIZATION returns one randomly.

Once a (likely) faulty statement  $s$  is selected, Algorithm 2 applies multiple mutations within the loop in lines 6–9. First, one mutation is applied with probability  $p = 0.5^0 = 1$  using the routine APPLY-MUTATION (line 7). Then, a second mutation is applied with probability  $p = 0.5^1$ , a third mutation with probability  $p = 0.5^2 = 0.25$ , and so on. Therefore, in each iteration of the loop in lines 6–9, a mutation is applied with probability  $p = 0.5^{counter}$ , where  $counter + 1$  is the number of executed iterations. In the first iteration,  $p = 1$ , and hence, it is guaranteed that at least one mutation is performed. Such a formula has been previously used in the context of test case generation [14]. Section 3.2.2 describes the mutation operators used to generate candidate patches. These operators receive as input  $\pi$  and  $s$  produced by the fault localization step.

**3.2.2 Mutation Operators.** We define two search operators *modify* and *shift* based on the types of errors that can occur in integration



**Figure 4: Illustrating the shift operator: (a) selecting  $b_s$  and path  $\pi$ , and (b) applying the shift operator.**

rules of an ADS. *Modify* modifies the conditions in the non-leaf nodes while *shift* switches the order of the rules. The operators are defined below:

**Modify.** Let  $s$  be the decision statement (or non-leaf node) selected for mutation (based on its suspiciousness). As discussed in Section 2,  $s$  has the following form: `<if op1 operator threshold>`. The operator *modify* performs three types of changes: (1) changing a threshold value, (2) altering the direction of a relational operator (e.g.,  $\leq$  to  $\geq$ ), or (3) changing arithmetic operations (e.g.,  $+$  to  $-$ ). Note that the left operand  $op1$  may be an arithmetic expression involving some arithmetic operators and variables.

**Shift.** Given the path  $\pi \in \Pi$  executed at the time of the failure, the *shift* operator changes the order of rules in  $\pi$ . Let  $s \in \pi$  be the decision statement (or non-leaf node) selected using the fault localization formula. The mutation swaps  $s$  with a randomly selected node that either dominates  $s$  (i.e., it precedes  $s$  in  $\pi$ ) or is dominated by  $s$  (i.e., it succeeds  $s$  in  $\pi$ ). Dominators and post-dominators of a node  $s$  are highlighted in grey in Figure 4(a). Among all dominators and post-dominators, one node  $b_s$  is selected randomly. We identify two cases: (Case1)  $b_s$  is a dominator of  $s$ . In this case, the shift operator swaps  $b_s$  with  $s$  such that the former becomes a post-dominator of the latter. Further, the operator shifts up the sub-path from  $s$  to the leaf node in  $\pi$  (see Figure 4(b-1) for an illustration). This is because features activated based on the condition in node  $s$  may not be consistent with the condition in node  $b_s$  or vice versa. Therefore, the nodes  $s$  and  $b_s$  should be swapped together with the sub-paths connecting them to their leaves (i.e., to their corresponding features). (Case2)  $b_s$  is a post-dominator of  $s$ . The shift operator swaps  $b_s$  with  $s$ . As before, the sub-path from  $s$  to the leaf node of  $\pi$  is also shifted down with  $s$  (see Figure 4(b-2) for an illustration).

**3.2.3 Search Objectives.** In our approach, the search objectives are based on the failures exposed by the test suite  $TS$ . Each failing test  $tc \in TS$  exposes one (or more) failure(s) whose intensity is  $O(tc(u), r_j) \in [-1; 0]$ , where  $r_j$  is the violated safety requirements. Recall that smaller  $O(tc(u), r_j)$  values indicate more severe violations of  $r_j$ . Since ADS can have multiple failures/violations, the program repair problem can be formulated as a many-objective optimization problem, whose search objectives are:

$$\begin{cases} \max \Omega_1(\Pi) = \min_{tc \in TS} \{O(tc(u), r_1)\} \\ \dots \\ \max \Omega_k(\Pi) = \min_{tc \in TS} \{O(tc(u), r_k)\} \end{cases} \quad (3)$$

where  $\min_{tc \in TS} \{O(tc(u), r_i)\}$  correspond to the most severe failure among all  $tc \in TS$  for requirement  $r_i$ . The concept of optimality in many-objective optimization is based on the concept of *dominance* [32]. More precisely, a patch  $\Pi_1$  dominates another patch  $\Pi_2$

*if and only if*  $\Omega_i(\Pi_1) \geq \Omega_i(\Pi_2)$  for all  $\Omega_i$  and there exists one objective  $\Omega_j$  such that  $\Omega_j(\Pi_1) > \Omega_j(\Pi_2)$ . In other words,  $\Pi_1$  dominates  $\Pi_2$ , if  $\Pi_1$  is not worse than  $\Pi_2$  in all objectives, and it is strictly better than  $\Pi_2$  in at least one objective [32].

**3.2.4 Archive.** The archive stores the best partial fixes found during the search. At the beginning of the search (line 2 of Algorithm 1), the archive is initialized with the faulty rule set  $\Pi$ . Every time a new patch  $\Pi_o$  is created and evaluated, we compare it with all the patches stored in the archive. The comparison is based on the notion of dominance described in Section 3.2.3:

- If  $\Pi_o$  dominates one solution  $A$  in the archive,  $\Pi_o$  is added to the archive and  $A$  is removed from the archive.
- If no element in the archive dominates or is dominated by the new patch  $\Pi_o$ , the new patch is added to the archive as well.
- The archive remains unchanged if  $\Pi_o$  is dominated by one (or more) solution(s) stored in the archive.

Therefore, in each iteration, the archive is updated (lines 8 of Algorithm 1) such that it includes the non-dominated solutions (i.e., the best partial patches) found so far.

The number of non-dominated solutions stored in the archive can become extremely large. To avoid this *bloating effect*, we add an upper bound to the maximum size for the archive. In this paper, we set the size of the archive to  $2 \times k$ , where  $k$  is the number of safety requirements. If the size of the archive exceeds the upper-bound, the solutions in the archive are compared based on the aggregated fitness value computed as follows:

$$D(\Pi) = \sum_{tc \in TS} \left( \sum_{r_j \in SR} O(tc(u), r_j) \right)$$

Then, we update the archive by selecting the  $2 \times k$  patches that have the largest  $D$  values.

One possible limitation of the evolutionary algorithms is that they can get trapped in some local optimum (premature convergence) [18]. A proven strategy to handle this potential limitation is to restart (re-initialize) the search [18]. We implement the following restarting policy in ARIEL: First, ARIEL uses a counter to count the number of subsequent generations with no improvements in the search objectives (i.e., the archive is never updated). If the counter reaches a threshold of  $h$  generations, the search is restarted by removing all partial patches in the archive and re-initializing it with the original faulty rule-set  $\Pi$  (see Section 4.3 for the value of  $h$ ).

**3.2.5 Termination Criteria.** The search stops when the search budget is consumed or as soon as a test-adequate patch is found, i.e., the *archive* contains one single patch  $\Pi^*$  that satisfies all test cases  $tc \in TS$ . Note that even though our repair algorithm is multi-objective, it generates only one final solution. This is because our algorithm, being a single-state search, generates one solution in each iteration and stops if that solution is Pareto optimal, i.e., it dominates all partial patches in the archive.

Indeed, our repair strategy decomposes the single objective of finding a single patch that passes all test cases in  $TS$  into multiple sub-objectives of finding partial patches that pass a subset of test cases in  $TS$ . This search strategy is called *multi-objectivization* [17, 23] and is known to help promote diversity. As shown in prior work in numerical optimization, *multi-objectivization* can lead to better results than classical single objective approaches when solving/targeting the same optimization problem [23]. In our context,

multi-objectivization helps to store partial patches that individually fix different faults by saving them in the *archive* (elitism). Such partial patches can be further mutated in an attempt to combine multiple partial patches, thus, building up the final patch/solution that fixes all faults. As shown in our evaluation in Section 4, our multi-objective approach outperforms single-objective search.

**3.2.6 Patch Minimization.** At the end of the search, ARIEL returns a patch that is test-adequate (i.e., it passes all tests), but it may include spurious mutations that are not needed for the fix. This problem can be addressed through a patch minimization algorithm [27], i.e., a greedy algorithm that iteratively removes one mutation at a time and verifies whether the patch after the reversed change still passes the test suite. If so, a smaller (minimized) patch that is still test-adequate is obtained.

**3.2.7 How ARIEL Addresses O1-O4?** ARIEL includes several ingredients to address the challenges of repairing ADS integration rules described in Section 1. First, ARIEL applied many-objective search, where each objective corresponds to a different failure (**O2**). Through many-objective optimization, ARIEL can repair several failures simultaneously. **O1** and **O4** are addressed by our fault localization formula (Equation 1) that assigns the suspiciousness score based on (1) the paths covered at the time of each failure, and (2) the severity of failures (i.e., failing tests). In addition, our search objectives address **O4** by measuring how a generated patch affects the severity of the failures. Finally, EA(1+1) generates and evaluates only one patch (offspring) at a time rather than evolving a pool of patches (like in GP-based approaches), thus addressing **O3**.

## 4 EVALUATION

In this section, we evaluate our approach to repairing integration rules using industria ADS systems.

### 4.1 Research Questions

**RQ1:** *How effective is ARIEL in repairing integration faults?* We assess the effectiveness of ARIEL by applying it to industrial ADS systems with faulty integration rules.

**RQ2:** *How does ARIEL compare to baseline program repair approaches?* In this research question, we compare ARIEL with three baselines that we refer to as Random Search (RS), Random Search with Multiple Mutations (RS-MM) and Genetic Programming (GP). We compare ARIEL with the baselines by evaluating their effectiveness (the ability to repair faults) and efficiency (the time it takes to repair faults). Section 4.3 describes the details of the baselines and our rationale for selecting them.

We conclude our evaluation by reporting on the feedback we obtained from automotive engineers regarding the usefulness and effectiveness of ARIEL in repairing faults in integration rules.

### 4.2 Case Study Systems

In our experiments, we use two case study systems developed in collaboration with company A. These two systems contain the four self-driving features introduced in Section 2, but contain two different sets of integration rules to resolve conflicts among multiple maneuvers issued by these features. We refer to these systems as *AutoDrive1* and *AutoDrive2*. Their features and their integration

rules are implemented in the Matlab/Simulink language [1]. We use PreScan [39], a commercial simulation platform available for academic research to execute *AutoDrive1* and *AutoDrive2*. The PreScan simulator is also implemented in Matlab/Simulink and *AutoDrive1* and *AutoDrive2* can be easily integrated into PreScan.

*AutoDrive1* and *AutoDrive2* include a TIS (Technology Independent Sensor) [33] sensor to detect objects and to identify their position and speed and a camera. The perception layer receives the data and images from the sensor and the camera, and consists, among others, tracking algorithms [13] to predict the future trajectory of mobile objects and a machine learning component that is based on support vector machine (SVM) to detect and classify objects. Our case study systems both include a Simulink model with around 7k blocks and 700k lines of Matlab code capturing the simulator, the four ADS features and the integration rules. Each decision tree for the integration rules of *AutoDrive1* and *AutoDrive2* has 204 rules (branches), and the total clauses in the rules' preconditions is 278 for each case study system. The set of rules of these two systems vary in their order and their threshold values. Syntactic diff shows 30% overlap between the ordered rules of *AutoDrive1* and *AutoDrive2*. The matlab/Simulink models of our case studies and our implementation of ARIEL are available online [2] and are also submitted alongside the paper.

*AutoDrive1* and *AutoDrive2* also include test suites with nine and seven system-level test cases, respectively. The test suite for *AutoDrive1* contains four failing test cases, while the test suite for *AutoDrive2* has two failing tests. Each failing test case for *AutoDrive1* and *AutoDrive2* violates one of the safety requirements in Table 1. Each test case for *AutoDrive1* and *AutoDrive2* executes its respective ADS systems (including the rules) for 1200 simulation steps. The two system-level test suites have been manually designed by developers to achieve high structural coverage (i.e., 100% branch and statement coverage) on the code of the integration rules. The total test execution time for the test suites of *AutoDrive1* and *AutoDrive1* are 20 and 30 minutes, respectively. Furthermore, each test suite contains at least one passing test case that exercises exactly one of the five safety requirements shown in Table 1.

Note that each test case in the test suites of *AutoDrive1* and *AutoDrive2* executes the system for 1200 times. The test suites achieve full structural coverage over the integration rules while requiring 20min and 30min to execute. Hence, larger test suites were neither needed nor practical in our context. Further, *AutoDrive1* and *AutoDrive2* give rise to several complex feature interaction failures one of which is discussed in Section 4.4. As the results of RQ1 and RQ2 confirm, the feature interaction failures in our case study systems cannot be repaired by existing standard and baseline automated repair algorithms, demonstrating both the complexity of our case studies and the need for our approach, ARIEL.

### 4.3 Baseline Methods and Parameters

In this section, we describe our baseline methods: Random Search (RS), Random Search with Multiple Mutations (RS-MM) and Genetic Programming (GP) and the parameters used in our experiments.

Unlike ARIEL<sup>1</sup>, our baselines use one objective, *baselineO*, similar to that used in most existing repair approaches [28, 44] and defined as follows:  $\text{baselineO} = W_p \times \text{total\_passed} + W_f \times \text{total\_failed}$ , where  $W_p$  and  $W_f$  are weights applied to the total number passing and failing test cases, respectively. The weight  $W_f$  is typically much larger than the weight  $W_p$  since typically there are more passing test cases than the failing ones in a test suite.

**4.3.1 Random Search (RS).** RS is a natural baseline to compare with because prior work [37] showed that it is particularly effective in the context of program repair, often outperforming evolutionary algorithms. RS does not evolve candidate patches but generates random patches by mutating the original rules  $\Pi$  only once using either the *modify* or *shift* operator. The final solution (patch) among all randomly generated patches is the one with the best *baselineO* value.

**4.3.2 Random Search with Multiple Mutations (RS-MM).** RS-MM is an improved variant of RS where multiple mutations are applied to the faulty rule set rather than one single mutation as done in RS. In other words, RS-MM generates random patches using the same operator applied in ARIEL (Algorithm 2). Thus, with the second baseline, we aim to assess whether the differences (if any) between random search and ARIEL are due to the mutation operator or the evolutionary search we proposed in this paper.

**4.3.3 Genetic Programming (GP).** We apply GP to assess the impact of using a single-state many-objective search algorithm rather than classic genetic programming. Similar to the state-of-the-art automated repair algorithm [28], GP relies on genetic programming. It starts with a pool (population) of  $n$  randomly generated patches and evaluates them using the *baselineO* objective function, which is the single objective to optimize [28]. Instead, ARIEL evolves one single patch and starts with a singleton archive containing only  $\Pi$  (see line 2 in Algorithm 1). Further, at each iteration, GP generates  $n$  new offspring patches by applying the mutation operators to the parents patches. Then, it selects the best  $n$  elements among the new and old individuals to maintain constant the size of the population. Similar to ARIEL, GP applies mutation operators multiple times to each parent patch. Thus, GP and ARIEL share the same mutation operator (Algorithm 2). Note that GP does not use the crossover operator because the alternative integration rules in the population are incomparable (e.g., rules/trees with different roots and rules orderings).

**4.3.4 Parameters.** As suggested in the literature [28, 44], for the fitness function *baselineO* used in the three baseline methods RS, RS-MM, and GP, we set  $W_f = 10$  and  $W_p = 1$ . For GP, the recommended population size used in the literature is 40 [28, 44]. However, computing the fitness for a population of 40 individuals takes on average around 20 hours for our case studies (i.e., one iteration of GP takes around 20 hours with a population size of 40). Hence, both for our experiments and in practice, it will take a long time to run GP for several iterations with a large population size. Therefore, we set the (initial) population size to 10. Note that for ARIEL, as described in Section 3.2.4, the archive size is dynamically updated at

<sup>1</sup>Recall that ARIEL uses multiple objectives (i.e., one objective per each safety requirement) to select the best element from an archive (see Section 3.2.3).

$$\begin{array}{lcl}
 \text{rule3: } speed(t) > speed-limit \wedge \overline{(dist(c, sign(t)) < dist_{cs} + \alpha)} & \implies & \text{TSR.active}(t) \\
 \text{rule4: } speed(t) < speedLeadingCar(t) & \implies & \text{ACC.active}(t)
 \end{array}$$

**Figure 5: Resolving a feature interaction failure between TSR and ACC in the set of rules in Figure 2.** To resolve this failure, rule3 and rule4 in Figure 2 are swapped and the threshold value in the precondition of rule3 is modified.

each iteration and does not need to be set as a parameter. Based on some preliminary experiments, we set the parameter  $h$ , discussed in Section 3.2.4 to randomly restart the search, to eight.

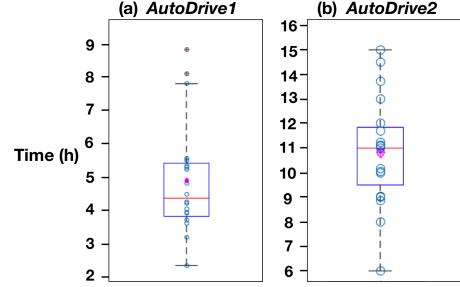
To account for the randomness of ARIEL, GP, RS and RS-MM, we reran each of them for 20 times on each case study system. For RQ1, we ran ARIEL on both case study systems until ARIEL finds a patch that passes all the test cases in the given test suite. We then used the maximum time needed by ARIEL to find patches as a timeout for the baseline methods in RQ2. We ran all the experiments on a laptop with a 2.5 GHz CPU and 16GB of memory.

#### 4.4 Results

**RQ1.** To answer this question, we apply ARIEL 20 times to *AutoDrive1* and *AutoDrive2* until a patch is found that passes all the test cases given as input. For both case study systems and for all the runs, ARIEL was able to repair the integration rules in the underlying ADS and terminate successfully. The box-plots in Figures 6(a) and (b) show the time needed for ARIEL to repair the integration rules of *AutoDrive1* and *AutoDrive2*, respectively, over 20 independent runs. As shown in the figures, ARIEL is able to repair all the integration faults in less than nine hours for *AutoDrive1* and in less than 16 hours for *AutoDrive2*. The average repair time for *AutoDrive1* and *AutoDrive2* is five hours and 11 hours, respectively.

We note that failures in *AutoDrive1-2* were caused by independent faults in different parts of the code. Recall from Section 3.2 that in order to resolve failures, ARIEL fixes the following fault types in the integration rules: (I) wrong operator/threshold, (II) wrong rule ordering and (III) a combination of both (I) and (II). For *AutoDrive1*, three faults were of type (I) and one was of type (III). For *AutoDrive2*, both faults were of type (II).

Figure 5 shows an example output of ARIEL when applied to the rules in Figure 2 to resolve the feature interaction failure between ACC and TSR exhibited by *AutoDrive1* and described in Section 2. In particular, the failure occurs when an ego car that is following a leading car with a speed higher than 50 reaches a traffic sign limiting the speed to 50. Since the rules in Figure 2 prioritize ACC over TSR, the car disregards the speed limit sign and continues with a speed above 50. The patch generated by ARIEL in Figure 5 swaps **rule3** and **rule4** in Figure 2, and in addition, modifies the threshold value for the pre-condition of the rule activating TSR. This ensures that TSR is applied before reaching the speed limit sign to be able to reduce the car speed below 50. Note that the passing test cases in the test suite of *AutoDrive1* will ensure ACC is still activated in scenarios when it should be activated.



**Figure 6: Time required for ARIEL to repair (a) *AutoDrive1* and (b) *AutoDrive2*.**

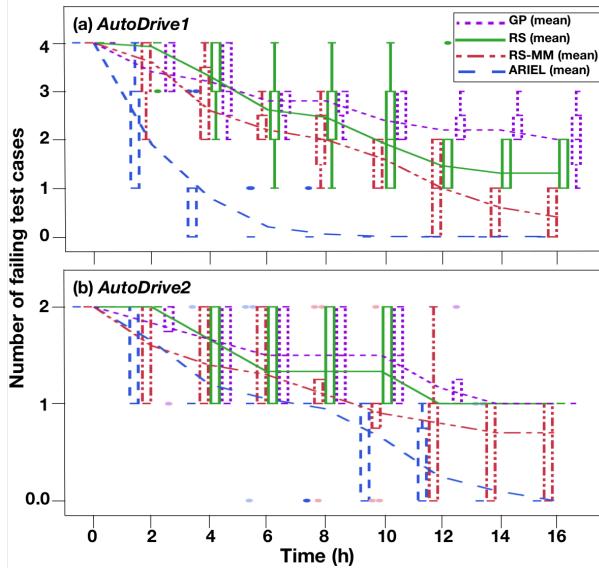
**Table 2: The results of applying GP, RS-MM, and RS to *AutoDrive1* and *AutoDrive2* for 16h.**

	GP (20 runs)	RS-MM (20 runs)	RS (20 runs)
<i>AutoDrive1</i> (4 failures)	- 6 runs fixed 1 failure - 11 runs fixed 2 failures - 3 runs fixed 3 failures	- 8 runs fixed 3 failures - 12 runs fixed 4 failures	- 6 runs fixed 2 failures - 14 runs fixed 3 failures
<i>AutoDrive2</i> (2 failures)	- 20 runs fixed 1 failure	- 15 runs fixed 1 failure - 5 runs fixed 2 failures	- 20 runs fixed 1 failure

The answer to RQ1 is that, on average, ARIEL is able to find correct and complete patches in five hours and 11 hours for *AutoDrive1* and *AutoDrive2*, respectively.

**RQ2.** We ran ARIEL, GP, RS-MM and RS 20 times for 16 hours. We selected a 16 hour timeout since it was the maximum time required by ARIEL to find patches for both *AutoDrive1* and *AutoDrive2*. Also, recall that we had four failing test cases for *AutoDrive1* and two failing test cases for *AutoDrive2*. Figures 7(a) and (b), respectively, show the number of failing test cases in *AutoDrive1* and *AutoDrive2* that are left unresolved over time by different runs of ARIEL, GP, RS-MM and RS. We show the results at every two-hour interval from 0 to 16h. As shown in Figures 7(a) and (b), all 20 runs of ARIEL are able to solve all the failing test cases in *AutoDrive1* after nine hours and in *AutoDrive2* after 16 hours. Note that this is consistent with the results shown in Figure 6. Table 2 reports the number of runs of GP, RS-MM, and RS, out of 20 runs, that resolved failures in *AutoDrive1* and in *AutoDrive2*. As shown in Figures 7(a) and (b) and in Table 2, none of the runs of GP and RS were able to resolve all the failures in *AutoDrive1* or in *AutoDrive2*. RS-MM had a better performance as some of its runs could fix all the failing test cases in our case studies.

As the above results show, GP had the worst performance in repairing the integration rules. This is because GP is a population-based search algorithm that evolves a pool of candidate patches iteratively. As a result, none of the GP runs could perform more than two iterations, hence its poor performance. Comparing RS and RS-MM shows that applying a sequence of mutations instead of a single mutation at each time is important in our context and helps the repair algorithms converge more quickly into a correct patch. Finally, ARIEL has the best performance compared to the



**Figure 7: Comparing the number of failing test cases obtained by ARIEL, RS-MM, RS and GP when applied.**

**Table 3:  $\hat{A}_{12}$  statistics obtained by comparing ARIEL with RS-MM, RS, and GP. ↑ indicates statistically significant results ( $p\text{-value} < 0.05$ ).**

Time	AutoDrive1			AutoDrive2		
	vs. RS-MM	vs. RS	vs. GP	vs. RS-MM	vs. RS	vs. GP
2h	↑ 0.03 (L)	↑ 0.04 (L)	↑ 0.04 (L)	0.26 (L)	↑ 0.04 (L)	0.12 (L)
4h	↑ 0.02 (L)	↑ 0.02 (L)	↑ 0.01 (L)	0.20 (L)	↑ 0.04 (L)	↑ 0.08 (L)
6h	0.00 (L)	↑ 0.01 (L)	↑ 0.00 (L)	0.19 (L)	0.05 (L)	↑ 0.08 (L)
8h	↑ 0.01 (L)	↑ 0.01 (L)	↑ 0.00 (L)	0.21 (L)	↑ 0.04 (L)	↑ 0.07 (L)
10h	↑ 0.00 (L)	↑ 0.00 (L)	↑ 0.00 (L)	0.19 (L)	0.03 (L)	↑ 0.04 (L)
12h	↑ 0.05 (L)	↑ 0.00 (L)	↑ 0.00 (L)	↑ 0.13 (L)	↑ 0.02 (L)	↑ 0.03 (L)
14h	↑ 0.05 (L)	↑ 0.00 (L)	↑ 0.00 (L)	↑ 0.10 (L)	↑ 0.01 (L)	↑ 0.02 (L)
16h	↑ 0.07 (L)	↑ 0.00 (L)	↑ 0.00 (L)	↑ 0.08 (L)	↑ 0.00 (L)	↑ 0.00 (L)

three baseline methods as all its runs find the desired patch within the 16h search time budget.

We compare the results in Figure 7 using a statistical test and effect sizes. Following existing guidelines [7], we use the non-parametric pairwise Wilcoxon rank sum test [10] and the Vargha-Delaney's  $\hat{A}_{12}$  effect size [41]. Table 3 reports the results obtained when comparing the number of failed test cases with ARIEL against GP, RS-MM and RS, when they were executed over time for AutoDrive1 and AutoDrive2. As shown in the table, for AutoDrive1, the p-values are all below 0.05 and the  $\hat{A}_{12}$  statistics show once again large effect sizes. For AutoDrive2, the p-values related to the results produced when the search time ranges between 12h and 16h are all below 0.05 and the  $\hat{A}_{12}$  statistics show large effect sizes. Hence, the number of failing test cases obtained by ARIEL when applied to AutoDrive1 and AutoDrive2 is significantly lower (with a large effect size) than those obtained by RS-MM, RS and GP.

The answer to RQ2 is that ARIEL significantly outperforms the baseline techniques for repairing integration faults when applied to our two case study systems.

**Feedback from domain experts.** We conclude our evaluation by reporting the feedback we received from engineers regarding the practical usefulness and benefits of our automated repair technique. In particular, we investigated whether the patched integration rules generated by our approach were understandable by engineers, and whether engineers could come up with such patches on their own using their domain knowledge and reasoning, all this without relying on our technique. The feedback we report here is based on the comments the lead engineers at company A (our partner company) made during two separate meetings. The engineers who participated in these meetings were involved in the development of AutoDrive1 and AutoDrive2 and were fully knowledgeable about the details of these two systems and the simulation platform used to test these systems. During the meetings, we discussed the four failing test cases of AutoDrive1, and the two failing test cases of AutoDrive2 that we had received from company A. Recall that as discussed in Section 4.3, these test cases were failing due to errors in the integration rules of AutoDrive1 and AutoDrive2. For each test case, we asked engineers whether they could think of ways to resolve the failure by suggesting modifications to the integration rules. Then, we presented to the engineers the patches generated by ARIEL. Note that we have alternative patches for each failing test case since we applied ARIEL several times to our case studies. We randomly selected two patches for each failure to be used in our discussions. We then asked the engineers whether they understood the automatically generated patches. We further discussed whether or not the automatically generated patches are the same as the fixes suggested by the engineers earlier in the meetings.

When at the beginning of the meetings, we asked engineers to propose patches based on their domain knowledge and reasoning and, in almost all cases, they proposed additional integration rules. Specifically, after reviewing the simulation of each failing test case, the engineers identified the conditions that had led to that specific failure. Then, they synthesized a new rule such that upon satisfaction of the conditions leading to the failure, the rule activates a feature that could avoid that failure. In contrast, ARIEL repairs integration faults either by modifying operators and thresholds in the clauses of the rules preconditions or by re-ordering the integration rules (see Section 3.2.2), and it never adds new rules.

In summary, the feedback we received from the engineers showed that they found the patches generated by ARIEL understandable and optimal. They agreed that adding new rules must be avoided if the integration faults can be resolved by modifying the existing rules since it is difficult to maintain a large number of rules. Further, as they add new rules they may introduce dead code or new faults in the system. The engineers further admitted that it is impossible to manually come up with resolutions that ARIEL can generate automatically since one cannot analyze the impact of varying thresholds, logical operators or rule reordering without automated assistance. Based on the feedback we received from the engineers, they concurred that the resolutions generated by ARIEL are valid, understandable, useful and optimal and they cannot be

produced by engineers. We note that ARIEL is not able to handle integration failures when the failure is due to a missing integration rule. In such cases, engineers can review the unaddressed failing test cases and try to manually synthesize rules that can handle such failures.

## 5 THREATS TO VALIDITY

The main threat to external validity is that our results may not generalize to other contexts. We distinguish two dimensions for external validity: (1) the applicability of our approach beyond our case study system, and (2) obtaining the same level of benefits as observed in our case study. As for the first dimension, we note that, in this paper, we provided two industrial ADS case studies. Our approach is dedicated to cases when there is an integration component deciding, based on rules, what commands are sent to actuators among alternative commands coming from different features, at every time step. In our context, as it is commonly done for cyber-physical systems, testing is done through simulation of the environment and hardware. The mutation operators used in our approach are general for integration rules in automated driving systems (the target of the paper) and for any cyber-physical systems where features send commands to common actuators and conflicts are prevented by a rule-based integration component (common in self-driving cars, robots, etc.). Our framework can be further extended with other operators if needed in different contexts. With respect to the second dimension, while our case study was performed in a representative industrial setting, additional case studies remain necessary to further validate our approach. In summary, our framework is generalizable to other domains with expensive test suites and similar integration architecture. If needed, mutation operators can be easily extended given the general structure of our framework.

## 6 RELATED WORK

We classified the related work (Table 4) by analyzing whether the existing automated repair approaches can handle multi-location faults (**C1**)? whether they can handle repairing systems that are expensive to execute (**C2**)? and whether they are able to distinguish between faults with different severity levels (**C3**)? As shown in the table, none of the existing repair techniques can address these three criteria, while as discussed earlier in the paper, ARIEL is designed under the assumption that test cases are expensive to run. Further, ARIEL can simultaneously repair multiple faults that might be multi-location and can prioritise repairing more severe faults over less severe ones. Below we discuss the related work included in Table 4 in more detail.

Many existing automated repair techniques rely on Genetic Programming (e.g., Arcuri et. al. [6, 8], GenProg [28], Marriagent [25], pyEDB[3], Par [22], and ARC [21]) and are similar to the GP baseline we used in Section 4 to compare with ARIEL. As shown in our evaluation, GP could not repair faults in our computationally expensive case studies (hence, failing **C2**). Among the techniques based on Genetic Programming, only ARC [21] is able to handle multi-location faults (**C1**) since it uses a set of pre-defined templates to generate patches. However, it does not address **C2** nor **C3**.

**Table 4: Classification of the related work based on the following criteria: Can the approach handle multi-location faults (**C1**)? Can the approach repair computationally expensive systems (**C2**)? Does the approach prioritise fixing more critical faults over the less critical ones (**C3**)?**

Ref	C1	C2	C3
[3, 22, 25, 28]	–	–	–
[21]	+	–	–
[5, 11, 12, 19, 35, 37, 38, 43]	–	+	–
[30, 31, 34, 42]	+	+	–

Several techniques use random search (e.g., RSRepair [37] and SCRepair [19]) to automate program repair. Although RSRepair [37] indicates that random search performs better than GenProg in terms of the number of iterations required for repair, a recent study [24] showed that GenProg performs better than RSRepair when applied to subjects different from those included in the original dataset of GenProg. We included two baselines based on random search in our evaluation (RS and RS-MM in Section 4). As shown there, while random search is more efficient than GP in repairing computationally expensive systems, it still underperforms ARIEL since it does not maintain an archive of partially repaired solutions. Further, as Table 4 shows, repair approaches based on random search do not address **C1** nor **C3**.

Similar to ARIEL, SPR [30], Prophet [31], AutoFix-E [42], and Angelix [34] can address both **C1** and **C2**. To handle multi-location faults, SPR [30], Prophet [31] and AutoFix-E [42] use pre-defined templates that are general repair solutions and are typically developed based on historical data or for the most recurring faults. In our context and at early stages of function modelling for ADS, we often do not have access to archival data and cannot develop such generic templates. Further, faults not conforming to some pre-defined templates may not be repaired if we only rely on template-induced patches. Angelix [34] uses constraint solving to synthesise multi-line fixes. Exhaustive and symbolic constraint solvers, however, face scalability issues and often are inapplicable to black-box simulation systems such as ADS simulators [4]. In contrast to these techniques, and as demonstrated by our evaluation, ARIEL succeeds in addressing **C1** to **C3** for complex simulation-based systems in the context of automated driving systems.

## 7 CONCLUSION

We proposed a repair technique to automatically resolve integration faults in automated driving systems (ADS). Our approach localizes faults over several lines of code to fix complex integration faults and deals with the scalability issues of testing and repairing ADS. Our repair algorithm relies on a many-objective, single-state search algorithm that uses an archive to keep track of partial repairs. Our approach is evaluated using two industrial ADS. The results indicate that our repair strategy can fix the integration faults in these two systems and outperforms existing automated repair techniques. Feedback from domain experts indicates that the patches generated by our approach are understandable by engineers and could not have been developed by them without any automation assistance.

## ACKNOWLEDGMENTS

This project has received funding from IEE S.A., Luxembourg, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), and NSERC of Canada under the Discovery and CRC programs.

## REFERENCES

- [1] 2019. Matlab/Simulink. <https://nl.mathworks.com/products/simulink.html>.
- [2] 2020. Appendix. <https://bitbucket.org/anonymous83/faultrepair/src/master/>. Also submitted along with the paper.
- [3] Thomas Ackling, Bradley Alexander, and Ian Grunert. 2011. Evolving Patches for Software Repair. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'11)* (Dublin, Ireland). ACM, New York, NY, USA, 1427–1434.
- [4] Rajeev Alur. 2015. *Principles of Cyber-Physical Systems*. MIT Press.
- [5] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the International Conference on Software Engineering (ICSE Companion'08)* (Leipzig, Germany). ACM, New York, NY, USA, 1003–1006.
- [6] Andrea Arcuri. 2011. Evolutionary Repair of Faulty Software. *Applied Software Computing* 11, 4 (June 2011), 3494–3514.
- [7] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.
- [8] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence (WCCI'08))*. IEEE, Hong Kong, 162–168.
- [9] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing Autonomous Cars for Feature Interaction Failures Using Many-objective Search. In *Proceedings of the International Conference on Automated Software Engineering (ASE'18)*. ACM, Montpellier, France, 143–154.
- [10] J. Anthony Capon. 1991. *Elementary Statistics for the Social Sciences: Study Guide*. Wadsworth Publishing Company, Belmont, CA, USA.
- [11] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09)*. IEEE, San Diego, CA, USA, 550–554.
- [12] Fávio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA'14)* (Hyderabad, India). ACM, New York, NY, USA, 30–39.
- [13] Eric Foxlin. 2005. Pedestrian tracking with shoe-mounted inertial sensors. *IEEE Computer graphics and applications* 25, 6 (2005), 38–46.
- [14] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
- [15] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.
- [16] John H. Holland. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.
- [17] Martin Jähne, Xiaodong Li, and Jürgen Branke. 2009. Evolutionary algorithms and multi-objectivization for the travelling salesman problem. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO'09)*. ACM, Montréal, Canada, 595–602.
- [18] Thomas Jansen. 2002. On the analysis of dynamic restart strategies for evolutionary algorithms. In *International Conference on Parallel Problem Solving from Nature (PPSN'02)*, Vol. 2. Springer, Granada, Spain, 33–43.
- [19] Tao Ji, Lijian Chen, Xiaoguang Mao, and Xin Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC'16)*, Vol. 1. IEEE, Atlanta, GA, USA, 197–202.
- [20] Wei Jin and Alessandro Orso. 2013. F3: fault localization for field failures. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, Lugano, Switzerland, 213–223.
- [21] David Kelk, Kevin Jallbert, and Jeremy S. Bradbury. 2013. Automatically Repairing Concurrency Bugs with ARC. In *Proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'13)*, João M. Lourenço and Eitan Farchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–84.
- [22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE, San Francisco, CA, USA, 802–811.
- [23] Joshua D Knowles, Richard A Watson, and David W Corne. 2001. Reducing local optima in single-objective problems by multi-objectivization. In *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO'01)*. Springer, Zurich, Switzerland, 269–283.
- [24] Xianglong Kong, Lingming Zhang, W Eri Wong, and Bixin Li. 2015. Experience report: How do techniques, programs, and tests impact automated program repair?. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE, Washington DC, USA, 194–204.
- [25] Ryotaro Kou, Yoshiki Higo, and Shinji Kusumoto. 2016. A Capable Crossover Technique on Automatic Program Repair. In *Proceedings of the International Workshop on Empirical Software Engineering in Practice (IWESEP'16)*. IEEE, Osaka, Japan, 45–50.
- [26] John R Koza and John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, Cambridge, MA, USA.
- [27] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each. In *Proceedings of the International Conference on Software Engineering (ICSE '12)* (Zurich, Switzerland). IEEE Press, Piscataway, NJ, USA, 3–13.
- [28] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (Jan 2012), 54–72.
- [29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.
- [30] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy). ACM, New York, NY, USA, 166–178. <https://doi.org/10.1145/2786805.2786811>
- [31] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)* (St. Petersburg, FL, USA). ACM, New York, NY, USA, 298–312. <https://doi.org/10.1145/2837614.2837617>
- [32] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu, Fairfax, Virginia, USA.
- [33] Ankith Manjunath, Ying Liu, Bernardo Henriques, and Armin Engstle. 2018. Radar Based Object Detection and Tracking for Autonomous Driving. In *Proceedings of the MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM'18)*. IEEE, Munich, Germany, 1–4.
- [34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'16)* (Austin, Texas). ACM, New York, NY, USA, 691–701.
- [35] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE '13)* (San Francisco, CA, USA). IEEE Press, Piscataway, NJ, USA, 772–781.
- [36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the International Conference on Software Engineering (ICSE '17)* (Buenos Aires, Argentina). IEEE Press, Piscataway, NJ, USA, 609–620. <https://doi.org/10.1109/ICSE.2017.62>
- [37] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. ACM, New York, USA, 254–265.
- [38] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)* (Baltimore, MD, USA). ACM, New York, NY, USA, 24–36.
- [39] TASS-International. 2019. PreScan. <https://www.tassinternational.com/prescan>.
- [40] Richard van der Horst and Jeroen Hogema. 1993. Time-to-collision and collision avoidance systems. In *Proceedings of the workshop of the International Cooperation on Theories and Concepts in Traffic Safety (ICTCT'93)*. Salzburg, Austria, 109–121.
- [41] András Varga and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
- [42] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated Fixing of Programs with Contracts. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'10)* (Trento, Italy). ACM, New York, NY, USA, 61–72. <https://doi.org/10.1145/1831708.1831716>
- [43] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*. IEEE, Silicon Valley, CA, USA, 356–366.
- [44] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*. IEEE, Vancouver, Canada, 364–374.

- [45] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4, Article 31 (Oct. 2013), 40 pages. <https://doi.org/10.1145/2522920.2522924>
- [46] Shin Yoo, Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, and Mark Harman. 2017. Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 1, Article 4 (June 2017), 30 pages. <https://doi.org/10.1145/3078840>

# CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair

Thibaud Lutellier  
tlutelli@uwaterloo.ca  
University of Waterloo  
Waterloo, ON, Canada

Yitong Li  
yitong.li@uwaterloo.ca  
University of Waterloo  
Waterloo, ON, Canada

Hung Viet Pham  
hvpham@uwaterloo.ca  
University of Waterloo  
Waterloo, ON, Canada

Moshi Wei  
m44wei@uwaterloo.ca  
University of Waterloo  
Waterloo, ON, Canada

Lawrence Pang  
lypang@edu.uwaterloo.ca  
University of Waterloo  
Waterloo, ON, Canada

Lin Tan  
lintan@purdue.edu  
Purdue University  
West Lafayette, IN, USA

## ABSTRACT

Automated generate-and-validate (G&V) program repair techniques (APR) typically rely on hard-coded rules, thus only fixing bugs following specific fix patterns. These rules require a significant amount of manual effort to discover and it is hard to adapt these rules to different programming languages.

To address these challenges, we propose a new G&V technique—CoCoNuT, which uses *ensemble* learning on the combination of convolutional neural networks (CNNs) and a new context-aware neural machine translation (NMT) architecture to automatically fix bugs in multiple programming languages. To better represent the context of a bug, we introduce a new context-aware NMT architecture that represents the buggy source code and its surrounding context separately. CoCoNuT uses CNNs instead of recurrent neural networks (RNNs), since CNN layers can be stacked to extract hierarchical features and better model source code at different granularity levels (e.g., statements and functions). In addition, CoCoNuT takes advantage of the randomness in hyperparameter tuning to build multiple models that fix different bugs and combines these models using ensemble learning to fix more bugs.

Our evaluation on six popular benchmarks for four programming languages (Java, C, Python, and JavaScript) shows that CoCoNuT correctly fixes (i.e., the first generated patch is semantically equivalent to the developer's patch) 509 bugs, including 309 bugs that are fixed by none of the 27 techniques with which we compare.

## CCS CONCEPTS

- Computing methodologies → Neural networks;
- Software and its engineering → Empirical software validation; Software defect analysis; Software testing and debugging.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397369>

## KEYWORDS

Automated program repair, Deep Learning, Neural Machine Translation, AI and Software Engineering

### ACM Reference Format:

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397369>

## 1 INTRODUCTION

To improve software reliability and increase engineering productivity, researchers have developed many approaches to fix software bugs automatically. One of the main approaches for automatic program repair is the G&V method [17, 38, 52, 69, 88, 89]. First, candidate patches are generated using a set of transformations or mutations (e.g., deleting a line or adding a clause). Second, these candidates are ranked and validated by compiling and running a given test suite. The G&V tool returns the highest-ranked fix that compiles and passes fault-revealing test cases.

While G&V techniques successfully fixed bugs in different datasets, a recent study [53] showed that very few correct patches are in the search spaces of state-of-the-art techniques, which puts an upper limit on the number of correct patches that a G&V technique can generate. Also, existing techniques require extensive customization to work across programming languages since most fix patterns need to be manually re-implemented to work for a different language.

Therefore, there is a need for a new APR technique that can fix more bugs (i.e., with a better search space) and is easily transferable to different programming languages.

Neural machine translation is a popular deep-learning (DL) approach that uses neural network architectures to generate likely sequences of tokens given an input sequence. NMT uses an *encoder* block (i.e., several layers of neurons) to create an intermediate representation of the input learned from training data and a *decoder* block to decode this representation into the target sequence. NMT has mainly been applied to natural language translation tasks (e.g., translating French to English).

APR can be seen as a translation from buggy to correct source code. Therefore, there is a unique opportunity to apply NMT techniques to learn from the readily available bug fixes in open-source repositories and generate fixes for unseen bugs.

Such NMT-based APR techniques have two key advantages. First, NMT models automatically learn complex relations between input and output sequences that are difficult to capture manually. Similarly, NMT models could also capture complex relations between buggy and clean code that are difficult for manually designed fix patterns to capture. Second, while G&V methods often use hard-coded fix patterns that are programming-language-dependent and require domain knowledge, NMT techniques can be retrained for different programming languages automatically without re-implementing the fix patterns, thus requiring little manual effort.

Despite the great potential, there are two main challenges of applying NMT to APR:

**(1) Representing context:** How to fix a bug often depends on the context, e.g., statements before and after the buggy lines. However, to represent the context effectively is a challenge for NMT models in both natural language tasks and bug fixing tasks; thus, the immediate context of the sentence to be translated is generally ignored. Two techniques that use NMT to repair bugs [13, 77] concatenate context and buggy code as one input instance. This design choice is problematic. First, concatenating context and buggy code makes the input sequences very long, and existing NMT architectures are known to struggle with long input. As a result, such approaches only fix short methods. For example, Tufano et al. [77] have to focus on short methods that contain fewer than 50 tokens. Second, concatenating buggy and context lines makes it more difficult for NMT to extract meaningful relations between tokens. For example, if the input consists of 1 buggy line and 9 lines of context, the 9 context lines will add noise to the buggy line and prevent the network from learning useful relations between the buggy line and the fix, which makes the models inefficient and less effective on fixing the buggy code.

We propose a new **context-aware NMT architecture** that has two separate encoders: one for the buggy lines, the other one for the context. Using separate encoders presents three advantages. First, the buggy line encoder will only have shorter input sequences. Thus, it will be able to extract strong relations between tokens in the buggy lines and the correct fix without wasting its resources on relations between tokens in the context. Second, a separate context encoder helps the model learn useful relations from the context (such as potential donor code, variables in scope, etc.) without adding noise to the relations learned from the buggy lines. Third, since the two encoders are independent, the context and buggy line encoders can have different complexity (e.g., different number of layers), which could improve the performance of the model. For example, since the context is typically much longer than the buggy lines, the context encoder may need larger convolution kernels to capture long-distance relations, while the buggy line encoder may benefit from a higher number of layers (i.e., a deeper network) to capture more complex relations between buggy and clean lines. *This context-aware NMT architecture is novel and can also be applied to improve other tasks such as fixing grammar mistakes and natural language translation.*

```
- catch (org.mockito.exceptionsverification.junit.  
       ArgumentsAreDifferent e) {  
+ catch (AssertionError e) {  
    (a) Patch for Mockito 5 in Defects4J for Java.  
  
    if (actualTypeArgument instanceof WildcardType) {  
        contextualActualTypeParameters.put(typeParameter, boundsOf((  
            WildcardType) actualTypeArgument));  
- } else {  
+ } else if (typeParameter != actualTypeArgument) {  
    contextualActualTypeParameters.put(typeParameter,  
        actualTypeArgument);  
}  
  (b) Patch for Mockito 8 only fixed by a context-aware model.
```

Figure 1: Two bugs in Defects4J fixed by CoCoNuT that other tools did not fix.

**(2) Capturing the diversity of bug fixes:** Due to the diversity of bugs and fixes (i.e., many different types of bugs and fixes), a single NMT model using the “best” hyperparameters (e.g., number of layers) would struggle to generalize.

Thus, we leverage **ensemble learning** to combine models of different levels of complexity that capture different relations between buggy and clean code. This allows our technique to learn diverse repair strategies to fix different types of bugs.

In this paper, we propose a new G&V technique called **CoCoNuT** that consists of an ensemble of fully convolutional (FConv) models and new context-aware NMT models of different levels of complexity. Each context-aware model captures different information about the repair operations and their context using two separate encoders (one for buggy lines and one for the context). Combining such models allows CoCoNuT to learn diverse repair strategies that are used to fix different types of bugs while overcoming the limitations of existing NMT approaches.

Evaluated against 27 APR techniques on six bug benchmarks in four programming languages, CoCoNuT fixes 509 bugs, 309 of which have not been fixed by any existing APR tools. Figure 1 shows two of such patches for bugs in Defects4J [31], demonstrating CoCoNuT’s capability of learning *new* and *complex* relations between buggy and clean code from the 3,241,966 instances in the Java training set. CoCoNuT generates the patch in Figure 1a using a **new pattern** (i.e., updating the Error type to be caught) that previous work [12, 27, 29, 35, 37, 38, 45–48, 50, 58, 67–69, 71, 82, 86, 87] did not discover. In the Figure, “-” denotes a line to be deleted that is taken as input by CoCoNuT, while “+” denotes the line generated by CoCoNuT and is identical to the developer’s patch. These previous techniques represent a **decade of APR research** that uses manually-designed Java fix patterns. This demonstrates that CoCoNuT complements existing APR approaches by automatically learning new fix patterns that existing techniques did not have, and automatically fixing bugs that existing techniques did not fix. To fix the bug in Figure 1b, CoCoNuT learns from the context lines the correct variables (`typeParameter` and `actualTypeArgument`) to be inserted in the conditional statement.

This paper makes the following contributions:

- A new **context-aware NMT** architecture that represents context and buggy input separately. This architecture is independent of our fully automated tool and could be applied to solve general

- problems in other domains where long-term context is necessary (e.g., grammatical error correction).
- The first application of *CNN* (i.e., FConv [20] architecture) for APR. We show that the FConv architecture outperforms LSTM and Transformer architectures when trained on the same dataset.
  - An *ensemble* approach that combines context-aware and FConv NMT models to better capture the diversity of bug fixes.
  - CoCoNuT is the first APR technique that is *easily portable* to different programming languages. With little manual effort, we applied CoCoNuT to four programming languages (Java, C, Python, and JavaScript), thanks to the use of NMT and new tokenization.
  - A thorough evaluation of CoCoNuT on six benchmarks in four programming languages. CoCoNuT fixes 509 bugs, 309 of which have not been fixed by existing APR tools.
  - A use of attention maps to explain why certain fixes are generated or not by CoCoNuT.

Artifacts are available<sup>1</sup>.

## 2 BACKGROUND AND TERMINOLOGY

**Terminology:** A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task. Each type of layer represents a specific low-level transformation (e.g., convolution, pooling) of the input data with specific *parameters* (i.e., *weights*). We call DL *architecture* an abstraction of a set of DL networks that have the same types and order of layers but do not specify the number and dimension of layers. A set of *hyperparameters* specifies how one consolidates an architecture to a network (e.g., it defines the convolutional layer dimensions or the number of convolutions in the layer group). The hyperparameters also determine which optimizer is used in training along with the optimization parameters, such as the learning rate and momentum. We call a *model* (or trained model), a network that has been trained, which has fixed weights.

**Attention:** The attention mechanism [9] is a recent DL improvement. It helps a neural network to focus on the most important features. Traditionally, only the latest hidden states of the encoder are fed to the decoder. If the input sequence is too long, some information regarding the early tokens are lost, even when using LSTM nodes [9]. The attention mechanism overcomes this issue by storing these long-distance dependencies in a separate attention map and feeding them to the decoder at each time step.

**Candidate, Plausible, and Correct Patches:** We call patches generated by a tool *candidate patches*. Candidate patches that pass the fault triggering test cases are *plausible patches*. Plausible patches that are semantically equivalent to the developers' patches are *correct patches*.

## 3 APPROACH

Our technique contains three stages: training, inference, and validation. Figure 2 shows an overview of CoCoNuT. In the training phase, we extract tuples of buggy, context, and fixed lines from open-source projects. Then, we preprocess these lines to obtain sequences of tokens, feed the sequences to an NMT network, and tune the network with different sets of hyperparameters. We further

train the top-k models until convergence to obtain an ensemble of k models. Since each model has different hyperparameters, each model learns different information that helps fix different bugs.

In the inference phase, a user inputs a buggy line and its context into CoCoNuT. It then tokenizes the input and feeds it to the top-k best models, which each outputs a list of patches. These patches are then ranked and validated by compiling the patched project. CoCoNuT then runs the test suite on the compilable fixes to filter incorrect patches. The final output is a list of *candidate patches* that pass the validation stage.

Section 3.1 presents the challenges of using NMT to automatically fix bugs, while the rest of Section 3 describes the different components of CoCoNuT.

### 3.1 Challenges

In addition to the two main challenges discussed in Introduction, i.e., (1) **representing context** and (2) **capturing the diversity of fix patterns**, using NMT for APR has additional challenges:

(3) **Choice of Layers of Neurons:** While natural language text is read sequentially from left to right, source code is generally not executed in the same sequential way (e.g., conditional blocks might be skipped in the execution). Thus, relevant information can be located farther away from the buggy location (e.g., a variable definition can be lines away from its buggy use). As a result, traditional recurrent neural networks (RNN) using LSTM layers [56] may not perform well for APR.

To address these challenges, we build our new context-aware NMT architecture using convolutional layers as the main component of the two encoders and the decoder, as they better capture such different dependencies than RNN layers [16, 20]. We stack convolutional layers with different kernel sizes to represent relations of different levels of granularity. Layers of larger kernel sizes model long-term relations (e.g., relations within a function), while layers of smaller kernel sizes model short-term relations (e.g., relations within a statement). To further track long-term dependencies in both input and context encoders, we use a multi-step attention mechanism.

(4) **Large vocabulary size:** Compared to traditional natural language processing (NLP) tasks such as translation, the vocabulary size of source code is larger and many tokens are infrequent because developers can practically create arbitrary tokens. In addition, letter case indicates important meanings in source code (e.g., zone is a variable and ZONE a constant), which increases the vocabulary size further. For example, previous work [13] had to handle a code vocabulary size larger than 560,000 tokens. Practitioners need to cut the vocabulary size significantly to make it scalable for NMT, which leaves a large number of infrequent out-of-vocabulary tokens. We address this challenge by using a new tokenization approach that reduces the vocabulary size significantly without increasing the number of out-of-vocabulary words. For Java, our tokenization reduces the vocabulary size from 1,136,767 to 139,423 tokens while keeping the percentage of tokens out-of-vocabulary in our test sets below 2% (Section 3.3).

<sup>1</sup><https://github.com/lin-tan/CoCoNut-Artifact>

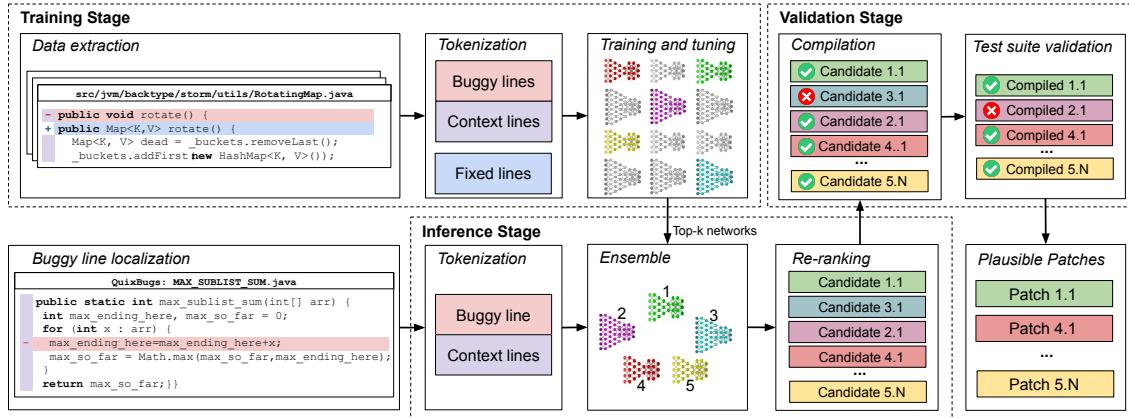


Figure 2: CoCoNuT's overview

### 3.2 Data Extraction

We train CoCoNuT on tuples of buggy, context, and fixed lines of code extracted from the commit history of open-source projects. To remove commits that are not related to bug fixes, we follow previous work [81] and only keep commits that have the keywords “fix,” “bug,” or “patch” in their commit messages. We also filter commits using six commit messages anti-patterns: “rename,” “clean up,” “refactor,” “merge,” “misspelling,” and “compiler warning.” We manually investigate a random sample of 100 commits filtered using this approach and 93 of them are bug-related commits. This is a reasonable amount of noise (7%) for ML training on large training data [34, 85]. We split commits into *hunks* (groups of consecutive differing lines) and consider each hunk as a unique *instance*.

We represent the context of the bug using the function surrounding the buggy lines because it gives semantic information about the buggy functionality, is relatively small, contains most of the relevant variables, and can be extracted for millions of instances. The block “Data Extraction” in Figure 2 shows an example with one buggy line (highlighted in red), one correct line (highlighted in blue), and some lines of context (with purple on the left).

The proposed context-aware NMT architecture is independent of the choice of the context and would still work with different context definitions (e.g., full file or data-flow context). Our contribution is to represent the buggy line and the context separately as a second encoder, independently of the chosen context. Exploring other context definitions is interesting future work.

### 3.3 Input Representation and Tokenization

CoCoNuT has two separate inputs: a buggy line and its context. The block “Buggy line localization” in Figure 2 shows the inputs for the MAX\_SUBLIST\_SUM bug in QuixBugs [43]. The line highlighted in red is the buggy line and the lines with a purple block on the left (i.e., the entire function) represent the context. Since typical NMT approaches take a vector of tokens as input, our first challenge is to choose a correct abstraction to transform a buggy line and its context into sequences of tokens.

We use a tokenization method analogous to word-level tokenization (i.e., space-separated), a widely used tokenization in NLP. However, word-level tokenization presents challenges that are specific to programming languages. First, we separate operators from variables as they might not be space-separated. Second, the vocabulary size is extremely large and many words are infrequent or composed of multiple words without separation (e.g., getNumber and get\_Number are two different words). To address this issue, we enhance the word-level tokenization by also considering underscores, camel letters, and numbers as separators. Because we need to correctly regenerate source code from the list of tokens generated by the NMT model, we also need to introduce a new token (<CAMEL>) to mark where the camel case split occurs. In addition, we abstract string and number literals except for the most frequent numbers (0 and 1) in our training set.

Thanks to these improvements, we reduce the size of the vocabulary significantly (e.g., from 1,136,767 to 139,423 tokens for Java), while limiting the number of out-of-vocabulary tokens (in our benchmarks, less than 2% of the tokens were out-of-vocabulary).

### 3.4 Context-Aware NMT Architecture

CoCoNuT’s architecture presents two main novelties. The first one consists of using two separate encoders to represent the context and the buggy lines (Figure 3). The second is a new application of fully convolutional layers (FConv) [20] for APR instead of traditional RNN such as LSTM layers used in previous work [13]. We choose the FConv layers because FConv’s CNN layers can be stacked to extract hierarchical features for larger contexts [16], which enables modeling source code at different granularity levels (e.g., variable, statement, block, and function). This is closer to how developers read code (e.g., looking at the entire function, a specific block, and then variables). RNN layers model code as a sequence, which is more similar to reading code from left to right.

We evaluate the impact of FConv and LSTM in Section 5.3.

Our architecture consists of several components: an input encoder, a context encoder, a merger, a decoder, and an attention module. For simplicity, Figure 3 only displays a network with one

convolutional layer. In practice, depending on the hyperparameters, a complete network has 2 to 10 convolutional layers for each encoder and the decoder.

In training mode, the context-aware model has access to the buggy lines, its context, and the fixed lines. The model is trained to generate the best representation of the transformation from buggy lines with context to fixed lines. In practice, this is conducted by finding the best combination of weights that translates the input instances from the training set to fixed lines. Multiple passes on the training data are necessary to obtain the best set of weights.

In inference mode, since the model does not have access to the fixed line, the decoder processes tokens one by one, starting with a generic <START> token. The outputs of the decoder and the merger are then combined through the multi-step attention module. Finally, new tokens are generated based on the outputs of the attention, the encoders, and the decoder. The generated token is then fed back to the decoder until the <END> token is generated.

Following the example input in Figure 3, a user inputs the buggy statement “`int sum=0;`” and its context to CoCoNuT. After tokenization, the buggy statement (highlighted in red) is fed to the Input encoder while the context (highlighted in purple) is fed to the Context encoder. The outputs of both encoders are then concatenated in the Merger layer.

Since CoCoNuT did not generate any token yet, the token generation starts by feeding the token <START> to the decoder (iteration 0). The output of the decoder ( $d_{out}$ ) is then combined with the merger output using a dot product to form the first column of the attention map. The colors of the attention map indicate how important each input token is for generating the output. For example, to generate the first token (double), the token `int` is the most important input token as it appears in red. The token generation combines the output of the attention, as well as the sum of the merger and decoder outputs ( $\Sigma e_{out}$  and  $\Sigma d_{out}$ ) to generate the token double. This new token is added to the list of generated tokens and the list is given back as a new input to the decoder (iteration 1). The decoder uses this new input to compute the next  $d_{out}$  that is used to build the second column of the attention map and to generate the next token. The token generation continues until <END> is generated.

We describe the different modules of the network below.

**Encoders and Decoder:** The purpose of the encoders is to provide a fixed-length vectorized representation of the input sequence while the decoder translates such representation to the target sequence (i.e., the patched line). Both modules have a similar structure that consists of three main blocks: an embedding layer, several convolutional layers, and a layer of gated linear units (GLU).

The embedding layers represent input and target tokens as vectors, with tokens occurring in similar contexts having a similar vector representation. In a sense, these layers represent the model’s knowledge of the programming language.

The output of the embedding layers is then fed to several convolutional layers. The size of the convolution kernel represents the number of surrounding tokens that are taken into consideration. Stacking such convolutional layers with different kernel sizes provides multiple levels of abstraction for our network to work with. For example, a layer with a small kernel size will only focus on a few surrounding tokens within a statement, while larger layers will

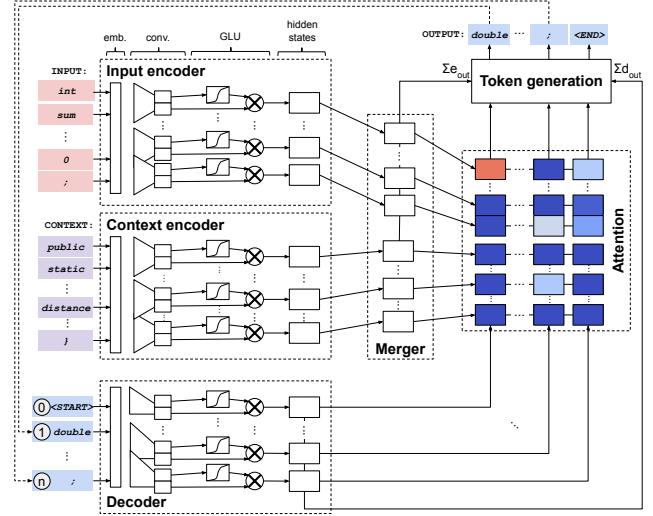


Figure 3: The new NMT architecture used in CoCoNuT.

focus on a block of code or even a full function. These layers are different in the encoder and the decoder. The encoders use information from both the previous and the next tokens in the input sequence (since the full input sequence is known at all times), while the decoder only uses information about the previously generated tokens (since the next tokens have not been generated yet). Figure 3 shows these differences (full triangles in the encoders and half triangles in the decoder).

After the convolutional layers, a layer of GLU (represented by the sigmoid and multiplication boxes in Figure 3) decides which information should be kept by the network. The Merger then concatenates the outputs of the input and context encoders.

**Multi-Step Attention:** Compared to traditional attention (see Section 2), multi-step attention uses an attention mechanism to connect the output of each convolutional layer in the encoders and the decoder. When multiple convolutional layers are used, it results in multiple attention maps. Multi-step attention is useful because it connects each level of abstraction (i.e., convolutional layer) to the output. This increases the amount of information an encoder passes to the decoder when generating the target tokens.

The attention map represents the impact of input tokens on generating a specific output token and can help explain why a specific output is generated. We analyze attention maps in RQ4.

**Token Generation:** The token generation combines the outputs of the attention layers, merger ( $\Sigma e_{out}$ ) and decoder ( $\Sigma d_{out}$ ) to generate the next token. Each token in the vocabulary is ranked by the token generation component based on their likelihood of being the next token in the output sequence. The token selected by the search algorithm is then appended to the list of generated tokens, and the list is sent back as the new input of the decoder. The token generation stops when the <END> token is generated.

**Beam Search:** We use a common search strategy called beam search to generate and rank a large number of candidate patches. For each iteration, the beam search algorithm checks the  $t$  most likely tokens ( $t$  corresponds to the beam width) and ranks them by the total likelihood score of the next  $s$  prediction steps ( $s$  correspond-

**Table 1: Training set information.**

Language	# projects	# instances
Java 2006	45,180	3,241,966
Java 2010	59,237	14,796,149
Python 2010	13,899	480,777
C 2005	12,577	2,735,506
JavaScript 2010	10,163	3,217,093

to the search depth). In the end, the beam search algorithm outputs the top  $t$  most likely sequences ordered based on the likelihood of each sequence.

### 3.5 Ensemble Learning

Fixing bugs is a complex task because there are very diverse bugs with very different fixing patterns that vary in terms of complexity. Some fix patterns are very simple (e.g., changing the operator  $<$  to  $>$ ) while others require more complex modifications (e.g., adding a null checker or calling a different function). Training a model to fix all types of bugs is difficult. Instead, it is more effective to combine multiple specialized models into an ensemble model that will fix more bugs than one single model.

Therefore, we propose an ensemble approach that combines: (1) models with and without context, and (2) models with different hyperparameters (different networks) that perform the best on our validation set.

As described in Section 2, hyperparameters consolidate an architecture to a network. Different hyperparameters have a large impact on the complexity of a network, the speed of the training process, and the final performance of the trained model. For this tuning process, we apply random search because previous work showed that it is an inexpensive method that performs better than other common hyperparameter tuning strategies such as grid search and manual search [10]. For each hyperparameter, based on our hardware limitations, we define a range from which we can pick a random value. Since training a model until convergence is very expensive, we tune with only one epoch (i.e., one pass on the training data). We train  $n$  models with different random sets of parameters to obtain models with different behavior and keep the top  $k$  best models based on the performance of each model on a separate validation set. This tuning process allows us to discard underfit or overfit models while keeping the  $k$  best models that converge to different local optima and fix different real-world bugs.

Finally, we sort patches generated by different models based on their ranks. We use the likelihood of each sequence (i.e., bug fix) generated by each model to sort patches of equal ranks.

### 3.6 Patch Validation

**Statement Reconstruction:** A model outputs a list of tokens that forms a fix for the input buggy line. The statement reconstruction module generates a complete patch from the list of tokens. For the abstracted tokens (i.e., strings and numbers), we extract donor code from the original file in which the bug occurred. Once the fix is generated, it is inserted at the buggy location, and we move to the validation step.

**Compilation and Test Suite Validation:** The model does not have access to the entire project; therefore, it does not know whether the generated patches are compilable or pass the test suite. The validation step filters out patches that do not compile or do not pass the triggering test cases. We use the same two criteria as previous work [89] for the validation process. First, the test cases that make the buggy version pass should still pass on the patched version. Second, at least one test case that failed on the buggy version should pass on the patched version.

### 3.7 Generalization to Other Languages

Since CoCoNuT learns patterns automatically instead of relying on handcrafted patterns, it can be generalized to other programming languages with minimum effort. The main required change is to obtain new input data. Fortunately, this is easy to do since our training set is extracted from open-source projects. Once the data for the new programming language has been extracted, the top  $k$  models can be retrained without re-implementation. CoCoNuT will learn fix patterns automatically for the new programming language.

## 4 EXPERIMENTAL SETUP

**Selecting Training Data:** Since the earliest bugs in our Java benchmarks are from 2006 and the latest are from 2016, we divide the instances extracted from the training projects into two training sets. The first one contains all instances committed before 2006 and the second one contains instances committed before 2010. Instances committed after 2010 are discarded. The models trained using the first training set can be used to train models to fix all bugs in the test benchmarks, while the models trained using the second training set should only be used on the bugs fixed after 2010. This setup is to ensure the validity of experiments so that no future data is used [75] (Section 5).

For Java, we use GHTorrent [21] to collect instances from 59,237 projects that contain commits before 2010. We also extract data from the oldest 20,000 Gitlab projects (restriction caused by the limitation of Gitlab search API) because we expect older projects to contain more bugs fixed before the first bug in our benchmarks, and 10,111 Bitbucket repositories (ranked by popularity because of limitations of the Bitbucket search API).

For other languages, we use GHTorrent to extract data from all GitHub projects before the first bug in their associated benchmark. Table 1 shows the number of projects and the number of instances in all training sets.

**Selecting State-of-the-art Tools for Comparison:** We compare CoCoNuT against 27 APR techniques, including all Java tools used in previous comparisons [47], two additional recent techniques [33, 71], five state-of-the-art C tools [2, 52, 55, 59], and two NMT-based techniques [13, 42]. We chose DLFix [42] and SequenceR [13] over other NMT techniques [23, 60, 77] for comparison, because [77] only generates templates and [23, 60] focus on compilation errors, which is a different problem.

**Training, Tuning, and Inference:** For tuning, we pick a random sample of 20,000 instances as our validation dataset and use the rest for training.

We use random search to tune hyperparameters. We limit the search space to reasonable values: embedding size (50-500), convolutional layer dimensions (128\*(1-5), (1-10)), number of convolutional layers (1-10), dropout, gradient clipping level, and learning rate (0-1). For tuning, we train 100 models for one epoch on the 2006 Java training set with different hyperparameters and rank the hyperparameters sets based on their perplexity [28], which is a standard metric in NLP that measures how well a model generates a sequence. We train the top-k (default k=10) models using ReduceLROnPlateau schedule, with a plateau of  $10^{-4}$ , and stop at convergence or until we reach 20 epochs. In inference mode, we use beam search with a beam width of 1,000.

**Infrastructure:** We use the Pytorch [65] implementations of LSTM, Transformer, and FConv provided by fairseq-py [1]. We train and evaluate CoCoNuT on three 56-core servers with NVIDIA TITAN V, Xp, and 2080 Ti GPUs.

## 5 EVALUATION AND RESULTS

**Realistic Evaluation Setup:** To evaluate CoCoNuT, we use six benchmarks commonly used for APR that contain realistic bugs.

When dealing with time-ordered data, it is not uncommon to incorrectly set up the evaluation [75]. If the historical data used to build the APR technique is more recent than the bugs in the benchmarks, it could contain helpful information (e.g., code clones and regression bugs) that would realistically be unavailable at the time of the fix. Using training/validation instances that are newer than the bugs in the benchmark to train or validate our models would be an incorrect setup and might artificially improve the performances of the models.

This incorrect setup potentially affects all previous APR techniques that use historical data. Although the effect may be smaller, even pattern-based techniques could suffer from this problem since patterns might have been manually learned from bugs that were fixed after the bugs in the benchmarks. To the best of our knowledge, we are the first to acknowledge and address this incorrect setup issue in the context of APR. The using-future-data threat is also one of the reasons that we did not use k-fold cross-validation for evaluation.

To address the setup issue, we extract the timestamp of the oldest bug in the benchmark (based on the date of the fixing commits) and only use training and validation instances before that timestamp. However, adding newer data to the training set would help CoCoNuT fix more recent bugs (e.g., using data until 2010 would be helpful to fix bugs from 2011). A straightforward solution would be to retrain CoCoNuT using different training data for each bug timestamp in the benchmark; however, this is not scalable. Instead, we split our benchmark into two parts. The first part contains bugs from 2006 to 2010 and is used to evaluate CoCoNuT trained with data from before 2006. The second part of the benchmark contains bugs from 2011 to 2016 and is used to evaluate CoCoNuT trained with data from before 2011 (including data from before 2006). This split allows CoCoNuT to learn from instances up to 2010 to fix newer bugs while keeping the overhead reasonable. We then combine the results of CoCoNuT on these two sub-benchmarks to obtain the final number of bugs fixed. With this correct setting, CoCoNuT has no access to data that would be unavailable in a realistic scenario.

Similar to previous work [12, 27, 29, 35, 38, 45–48, 50, 58, 67–69, 71, 82, 86, 87], we stop CoCoNuT after the first generated patch that is successfully validated against the test suite. If no patch passes the test suite after the limit of six hours, we stop and consider the bug not repaired by CoCoNuT. For evaluation purposes only, three co-authors manually compare the plausible patches (i.e., patches that pass the test cases) to the developers' patches and consider a patch *correct* if they all agree it is identical or semantically equivalent to the developers' patch using the equivalence rules described in previous work [49].

### 5.1 RQ1: How does CoCoNuT perform against state-of-the-art APR techniques?

**Approach:** Table 2 shows that we compare CoCoNuT with 27 state-of-the-art G&V approaches on six benchmarks for four different programming languages. We use Defects4J [31] and QuixBugs [43] for Java, Codeflaws [76] and ManyBugs [39] for C, and QuixBugs [43] for Python. For JavaScript; we use the 12 examples associated with common bug patterns in JavaScript described in previous work (BugAID) [25]. The total number of bugs in each dataset is under the name of the benchmark.

We compare our results with popular (e.g., GenProg) and recent (e.g., TBar) G&V techniques for C and Java. We do not compare with the JavaScript APR technique Vejovis [64] since it only fixes bugs in Document Object Model (DOM). We extract the results for each technique from a recent evaluation [46] and cross-check against original results when available. Perfect buggy location results are extracted from the GitHub repository of previous work [49] and manually verified to remove duplicate bugs.

We run Angelix, Prophet, SPR, and GenProg on the entire Codeflaws dataset because these techniques had not been evaluated on the full dataset. We use the default timeout values provided by the Codeflaws dataset authors. Since Codeflaws consists of small single-file programs, it is unlikely that these techniques would perform differently with a larger timeout.

The results in Table 2 are displayed as x/y, with x the number of correct patches that are ranked first by an APR technique, and y the number of plausible patches. We also show in parentheses the number of bugs fixed by CoCoNuT that have not been fixed by other techniques. ‘-’ indicates that a technique has not been evaluated on the benchmark or does not support the programming language of the benchmark. For the BugAID and ManyBugs benchmarks, we could not run the validation step; therefore we only display the number of patches that are identical to developers' patches ( $\dagger$  in the Table) and cannot show the number of plausible patches.

Since different approaches used different fault localization (FL) techniques, we separate them based on FL types (Column FL), as was done in previous work [48]. Standard FL-based approaches use a traditional spectrum-based fault localization technique. Supplemented FL-based APR techniques use additional methods or assumptions to improve FL. For example, HD-Repair assumes that the buggy file and method are known. Finally, Perfect FL-based techniques assume that the perfect localization of the bug is known. According to recent work [46, 49], this is the preferred way to evaluate G&V approaches, as it enables fair assessment of APR techniques independently of the fault localization approach used.

**Table 2: Comparison with state-of-the-art G&V approaches.** The number of bugs that only CoCoNuT fixes is in parentheses (309). The results are displayed as x/y, with x the number of bugs correctly fixed, and y the number of bugs with plausible patches. \* indicates tools whose manual fix patterns are used by TBar. Numbers are extracted from either the original papers or from previous work [49] which reran some approaches with perfect localization. In Defects4J, we exclude Closure 63 and Closure 93 from the total count since they are duplicates of Closure 62 and 92, respectively. <sup>†</sup> indicates the number of patches that are identical to developer patches—the minimal number of correct patches. - indicates tools that have not been evaluated on a specific benchmark. The highest number of correct patches for each benchmark is in bold.

FL	Tool	Java		C		Python		JavaScript	
		Defects4J 393 bugs	QuixBugs 40 bugs	Codeflaws 3,902 bugs	ManyBugs 69 bugs	QuixBugs 40 bugs	BugAID 12 bugs		
Standard	1 Angelix [59]	-	-	318/591	<b>18/39</b>	-	-		
	2 Prophet [55]	-	-	301/839	15/39	-	-		
	3 SPR [52]	-	-	283/783	11/38	-	-		
	4 Astor* [58]	-	6/11	-	-	-	-		
	5 LSRepair [45]	19/37	-	-	-	-	-		
	6 DLFix [42]	29/65	-	-	-	-	-		
Supplemented	7 JAID [12]	9/31	-	-	-	-	-		
	8 HD-Repair* [37]	13/23	-	-	-	-	-		
	9 SketchFix* [27]	19/26	-	-	-	-	-		
	10 ssFix* [86]	20/60	-	-	-	-	-		
	11 CapGen* [82]	21/25	-	-	-	-	-		
	12 ConFix [33]	22/92	-	-	-	-	-		
	13 Elixir* [69]	26/41	-	-	-	-	-		
	14 Hercules [71]	49/72	-	-	-	-	-		
	15 SOSRepair [2]	-	-	-	16/23	-	-		
	16 Nopol [88]	2/9	1/4	-	-	-	-		
	17 (j)Kali [58, 68]	2/8	1/2	-	3/27	-	-		
	18 (j)GenProg [38, 58]	6/16	0/2	[255–369]/1423	2/18	-	-		
	19 RSRepair [67]	10/24	2/4	-	2/10	-	-		
	20 ARJA [91]	12/36	-	-	-	-	-		
	21 SequenceR [13]	12/19	-	-	-	-	-		
Perfect	22 ACS [87]	16/21	-	-	-	-	-		
	23 SimFix* [29]	27/50	-	-	-	-	-		
	24 kPAR* [46]	29/56	-	-	-	-	-		
	25 AVATAR* [48]	29/50	-	-	-	-	-		
	26 FixMiner* [35]	34/62	-	-	-	-	-		
	27 TBar [47]	52/85	-	-	-	-	-		
CoCoNuT (not fixed by others)		44/85 (6)	13/ 20 (10)	423/ 716 (271)	7 †/- (0)	19/21 (19)	3 †/- (3)		
Total bugs fixed by CoCoNuT				509 (309)					

We exclude iFixR [36] because it uses kPAR to generate fixes which is already in the Table. We choose kPAR since its evaluation is similar to our evaluation setup and allows for a fairer comparison. iFixR proposes to use bug reports instead of test cases. However, we believe that it is reasonable to keep test cases for validation because, even if they were committed at a later stage, they were often available to the developers at the time of the bug report since developers generally discover bugs by seeing a program fail given one failing test case. Regardless, this is still a fair comparison with the 27 existing techniques which all use test cases for validation.

TBar is a recent pattern-based technique that uses patterns implemented by previous work (techniques marked with a \* in Table 2). *TBar shows how a combination of most existing pattern-based techniques behaves with perfect localization.*

**Results:** Overall, Table 2 shows that CoCoNuT fixes 509 bugs across six bug benchmarks in four programming languages. CoCoNuT is the best technique on four of the six benchmarks and is the only technique that fixes bugs in Python and JavaScript, indicating that

CoCoNuT is easily portable to different programming languages with little manual effort.

On the Java benchmarks, CoCoNuT outperforms existing tools on the QuixBugs benchmark, fixing 13 bugs, including 10 bugs that have not been fixed before. On Defects4J, CoCoNuT performs better than all techniques but TBar and Hercules. In addition, 6 bugs CoCoNuT fixes have not been fixed by any other techniques. Two of the 6 bugs require new fix patterns that have not been found by any previous work from a decade of APR research. We investigate these six bugs in detail in RQ2. In addition, fifteen of the bugs fixed by Hercules are multi-hunk bugs (i.e., bugs that need changes in multiple locations to be fixed), currently out of scope for CoCoNuT which mostly generates single-hunk bug fixes. In the future, the method used by Hercules to fix multi-hunk bugs could be applied to CoCoNuT.

Two of the 240 Defects4J bugs from after 2010 can only be fixed by models trained with the training set containing data up to 2010. Surprisingly, the models trained with data from before 2006 stay relevant and can fix bugs introduced 4 to 10 years after the training data. This highlights the fact that, while training DL

```
- static float toJavaVersionInt(String version) {
+ static int toJavaVersionInt(String version) {
```

**Figure 4: CoCoNuT’s Java Patch for Lang 29 in Defects4J that have not been fixed by other techniques.**

```
- int indexOfDot=namespace.indexOf('.');
+ int indexOfDot=namespace.lastIndexOf('.');
```

(a) Java patch for Closure 92 in Defects4J.

```
- int end = message.indexOf(templateEnd,start);
+ int end = message.lastIndexOf(templateEnd,start);
```

(b) Similar change to Closure 92 bug occurring in the training data.

**Figure 5: Example of patches fixed only by CoCoNuT and related changes in the training set.**

```
- while True:
+ while queue:
    (a) Python patch for BREADTH_FIRST_SEARCH in QuixBugs.
```

```
- while (true) {
+ while(!queue.isEmpty()) {
```

(b) Java patch for BREADTH\_FIRST\_SEARCH in QuixBugs.

**Figure 6: BREADTH\_FIRST\_SEARCH bugs fixed by CoCoNuT in both Python and Java QuixBugs benchmarks.**

models is expensive, such models stay relevant for a long time and do not need expensive retraining.

In Defects4J, 43% (19 out of 44) of the bugs fixed by CoCoNuT are not fixed by TBar (the best technique), hence CoCoNuT complements TBar very well. There are also several bugs fixed by TBar that CoCoNuT fails to fix. Seven of them are if statement insertions (e.g., null check), and three of them are fixes that move a statement to a different location. These bugs are difficult to fix for CoCoNuT because we targeted bugs that are fixed by modifying a statement, hence these two transformations do not appear in our training set. Fixing bugs by inserting if statements has been widely studied [47, 52, 87, 88]. Instead, we propose a new technique that fixes bugs that are more challenging for other techniques to fix.

While we generate 1,000 patches for each bug per model, correct patches are generally ranked very high. Ten of the 44 correct patches in Defects4J are ranked first by one of the models. The average rank of a correct patch is 63 and the median rank is 4. The worst rank of a correct patch is 728.

A potential threat to validity is that some tools use different fault localization techniques. While we cannot re-implement all existing tools using perfect localization (e.g., some tools are not publicly available), we try our best to mitigate this threat. First, we consider 13 tools that also use perfect localization for comparison, including TBar, which fixed the most number of bugs in Defects4J. Second, TBar uses fix patterns from 10 tools (marked with \* in Table 2) with perfect localization. Thus, although some of these 10 tools do not use perfect fault localization, we indirectly compare with these tools using perfect fault localization. CoCoNuT fixes 6 bugs that have not been fixed by existing tools including TBar.

On the C benchmarks, CoCoNuT fixes 430 bugs, 271 of which have not been fixed before. CoCoNuT outperforms existing techniques on the Codeflaws dataset, fixing 105 more bugs than Angelix

and has a 59% precision (423 out of 716). On the ManyBugs benchmark, CoCoNuT fixes seven bugs, outperforming GenProg, Kali, and RSRepair but is outperformed by other techniques.

We manually check for semantic equivalence for all generated patches except for GenProg on ManyBugs. Instead, we manually check a random sample of 310 out of the 1,423 plausible patches for Codeflaws generated by GenProg, because GenProg rewrites the program before patching it, e.g., replacing a for loop with a while loop, which makes it difficult to manually investigate all 1,423 plausible patches. The margin of error for this sample is 4% with a 95% confidence level, thus, Table 2 shows the projected range of the number of Codeflaws bugs that GenProg fixes.

On the **JavaScript** and **Python** bug benchmarks, CoCoNuT fixes three and 19 bugs respectively.

Since different bug benchmarks contain different distributions of different types of bugs, and different APR tools fix different types of bugs, it is important to evaluate new APR techniques on different benchmarks for a fair and comprehensive comparison. CoCoNuT is the best technique on four of the six benchmarks.

**Summary:** CoCoNuT is the first approach that has been successfully applied without major re-implementation to different programming languages, fixing **509 bugs** in six benchmarks for four popular programming languages, **309** of which that have not been fixed by existing work, including six in Defects4J, a dataset that has been heavily used to evaluate 22 other tools.

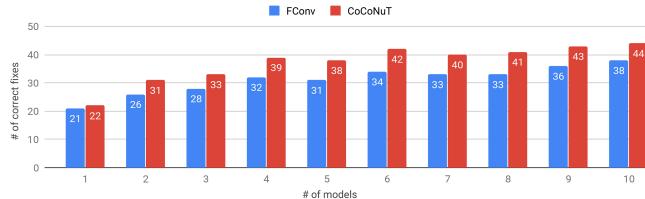
## 5.2 RQ2: Which bugs only CoCoNuT can fix?

**Approach:** For the bugs only CoCoNuT can fix in Defects4J, we rerun TBar, the best technique for Java, on our hardware, with perfect localization, and without a time limit to confirm that TBar cannot fix these bugs, even under the best possible conditions. For C, as stated in RQ1, we run Angelix, SPR, Prophet, and GenProg on Codeflaws with the same hardware we used for CoCoNuT for a fair comparison. CoCoNuT is the only technique fixing bugs in the Python and JavaScript benchmarks.

**Results:** CoCoNuT fixes bugs by automatically learning new patterns that have not yet been discovered, despite a decade of research on Java fix patterns. Mockito 5 (Figure 1a) is a bug in Defects4J only CoCoNuT fixes because it requires patterns that are not covered by existing pattern-based techniques (i.e., exception type update). Lang 29 (Figure 4) is another bug that cannot be fixed by other tools because existing pattern-based techniques such as TBar do not have a pattern for updating the return type of a function. TBar cannot generate any candidate patches for these two bugs.

Thanks to the context-aware architecture, CoCoNuT fixes bugs other techniques do not fix by correctly extracting donor code (e.g., correct variable names) from the context, while techniques such as TBar rely on heuristics. An example of such bugs is in Figure 1b as explained in Section 1.

CoCoNuT is the only technique that fixes Closure 92 because it learns from historical data (Figure 5b) that the `lastIndexOf` method in the `String` library is a likely candidate to replace the `indexOf` method. Other techniques such as TBar fail to generate a correct patch because the donor code is not in their search space.



**Figure 7: Number of bugs fixed as the number of models considered in ensemble learning increases.**

```
- Calendar c = new GregorianCalendar(mTimeZone);
+ Calendar c = new GregorianCalendar(mTimeZone, mLocale);
```

**Figure 8: CoCoNuT's Java patch for Lang 26 in Defects4J.**

CoCoNuT directly learns patterns from historical data without any additional manual work, making it portable to different programming languages. Figure 6 shows the **same** bug in both the Java (Figure 6a) and Python (Figure 6b) implementations of the QuixBugs dataset. CoCoNuT fixes both bugs, even if the patches in Java and Python are quite different. Adapting pattern-based techniques for Java to Python would require much work because the fix-patterns are very different, even for the same bug.

**Summary:** CoCoNuT fixes **309** bugs that have not been fixed by existing techniques, including six bugs in Defects4J. CoCoNuT fixes new bugs by (1) automatically learning **new patterns** that have not been found by previous work, (2) extracting donor code **from the context of the bug**, and (3) extracting donor code **from the historical training data**.

### 5.3 RQ3: What are the contributions of the different components of CoCoNuT and how does it compare to other NMT-based APR techniques?

**Approach:** To understand the impact of each component of CoCoNuT, we investigate them individually. More specifically, we focus on three key contributions: (1) the performance of our new NMT architecture compared to state-of-the-art NMT architectures, (2) the impact of context, and (3) the impact of ensemble learning. We compare CoCoNuT with DLFix [42] and SequenceR [13], two state-of-the-art NMT-based APR techniques and three other state-of-the-art NMT architectures (i.e., LSTM [56], Transformer [78], and FConv [20]). These models have not been used for program repair, so we implemented them in the same framework as our work (i.e., using Pytorch [65] and the fairseq-py [1] library). To ensure a fair comparison, we tune and train the LSTM, Transformer, and FConv similarly to CoCoNuT. SequenceR [13] uses an LSTM encoder-decoder approach to repair bugs automatically. We use the numbers reported by SequenceR and DLFix's authors on the Defects4J dataset for comparison since working versions of SequenceR and DLFix were unavailable at the time of writing. DLFix [42] uses a new treeRNN architecture that represents source code as a tree.

**Comparison with State-of-the-art NMT:** CoCoNuT fixes the most number of bugs, with 44 bugs fixed in Defects4J. DLFix is

the second best, fixing 29 bugs, followed by FConv with 21 bugs while Transformer and SequenceR have similar performances, fixing 13 and 12 bugs respectively. The last baseline, LSTM, performs poorly with only 5 bugs fixed. These results demonstrate that CoCoNuT performs better than state-of-the-art DL approaches and that directly applying out-of-the-box deep-learning techniques for APR is ineffective.

**Impact of the Context:** Figure 7 shows the total number of bugs fixed using the top-k models, with k from 1 to 10. For all k, CoCoNuT outperforms an ensemble of FConv models trained without using context. Our default CoCoNuT (with k = 10 models) fixes six more bugs than using models without context (44 versus 38).

**Advantage of Ensemble Learning:** Figure 7 shows that as k increases from 1 to 10, the number of bugs that CoCoNuT fixes increases from 22 to 44, a 50% improvement. We observe a similar trend for the FConv models, indicating that ensemble learning is beneficial independently of the architecture used. Model 9 and Model 7 are the best FConv and Context-aware models respectively, both fixing 26 bugs.

While increasing k also increases the runtime of the technique, this cost is not prohibitive because CoCoNuT is an offline technique and can be run overnight. In the worse case, with k=10, CoCoNuT takes an average of 16.7 minutes to generate 20,000 patches for one bug (for k=1, it takes an average of 1.8 minutes per bug).

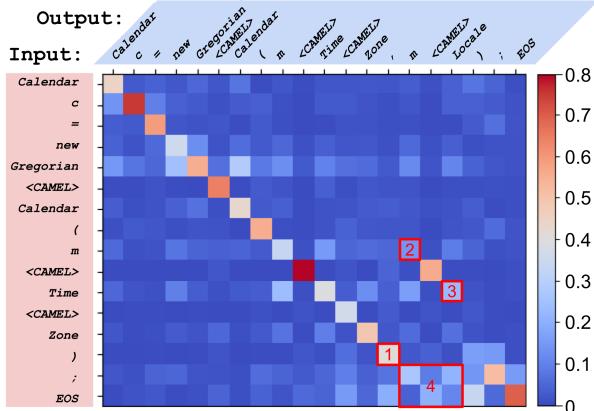
While CoCoNuT fixes 44 in Defects4J, the average number of bug fixed by a single model is 15.65. in Defects4J, 40% of the bugs are fixed by five or fewer models. Six of the correctly fixed bugs are only fixed by one model, while only two bugs are fixed by all models. This indicates that different models in our ensemble approach specialize in fixing different bugs.

**Summary:** The new NMT architecture we propose performs significantly better than baseline architectures. In addition, using ensemble learning to combine the models improves the results, with a 50% improvement for k=10 compared to k=1.

### 5.4 RQ4: Can we explain why CoCoNuT can (or fail to) generate specific fixes?

**The Majority of the Fixes are not Clones:** By learning from historical data, the depth of our neural network allows CoCoNuT to fix complex bugs, including the ones that require generating new variables. As discussed at the start of Section 5, we only keep training and validation instances from before the first bugs in our benchmarks for a fair evaluation. As a result, the exact same bug cannot appear in our training/validation sets and evaluation benchmarks. However, the same patch may still be used to fix different bugs introduced at different times in different locations. Having such patch clones in both training and test sets is valid, as recurring fixes are common. The majority of the bugs fixed by CoCoNuT do not appear in the training sets: only two patches from the C benchmark and one from the JavaScript benchmark appear in the training or validation sets. This indicates that CoCoNuT is effective in learning and generating completely different fixes.

**Analyzing the Attention Map:** CoCoNuT can also fix bugs that require complex changes. For example, the fix for *Lang 26* from



**Figure 9: Attention map for the correct patch of *Lang 26* from the Defects4J benchmark generated by CoCoNuT.**

the Defects4J dataset shown in Figure 8 requires injecting a new variable `mLocale`. This new variable only appears four times in our training set, and never in a similar context. However, `mLocale` contains the token `Locale` which co-occurs in our training set with the buggy statement tokens `Gregorian`, `Time`, and `Zone`.

The attention map in Figure 9 confirms that the token `Time` is important for generating the `Locale` variable. Specifically, the tokenized input is shown on the y-axis while the tokenized generated output is displayed on the x-axis. The token `<CAMEL>` between `m` and `Locale` indicates that these two tokens form one unique variable. The attention map shows the relationship between the input tokens (vertical axis) and the generated tokens (horizontal axis). The color in a cell represents the relationship of corresponding input and output tokens. The color scale, shown on the right of the figure, varies from 0 (dark blue) to 0.6 (dark red) and indicates the contribution of each input token for generating an output token.

This attention map helps us understand why the model generates specific tokens. For example, the second `m` in the output is generated because of the token `m` in the input (part of `mTimeZone`), showing that the network can keep the naming convention when generating new variables (square labeled 2 in Figure 9). The token `Locale` is mostly generated because of the token `Time`, indicating that the network is confident these tokens should be together (square 3). Finally, the tokens forming the variable `mLocale` are all influenced by the input tokens `)` and `;` and `EOS`, indicating that this token is often used right before the end of a statement (i.e., as the last parameter of the function call, rectangle 4 on Figure 9). This example shows how the attention map can be used to understand why CoCoNuT generates a specific patch.

**Limitations of Test Suites:** Patches that pass the test suite but are incorrect are an issue that CoCoNuT and other APR techniques share. CoCoNuT could generate a correct fix for 8 additional bugs if more time is given; however, for these 8 bugs, CoCoNuT generates an incorrect patch that passes the test suite (we only validate the first candidate patch due to time considerations) or timed out before the correct fix. Using enhanced test suites proposed in previous work [89] may alleviate this issue.

## 5.5 Execution Time

**Data Extraction:** Extracting data from open-source repositories and training CoCoNuT are one-time costs that can be amortized across many bugs and should not be counted in the end-to-end time to fix one bug. For Java, extracting data from 59,237 projects takes five days using three servers.

**Training Time:** The median time to train our context-aware NMT model for 1 epoch during tuning is 8.7 hours. On average, training a model for 20 epochs takes 175 hours on one GPU. Transformers and FConv networks are faster to train, taking an average of 2.5 and 2.7 hours per epoch. However, training the LSTM network is much slower (22 hours per epoch).

This one time cost is to be compared to **the decade of research** spent designing and implementing new fix patterns.

**Cost to fix one bug:** Once the model is trained, the main cost is the inference (i.e., generating patches) and the validation (i.e., running the test suite). During inference, generating 20,000 patches for one bug (CoCoNuT default setup) takes 16.7 min on average using one GPU. On our hardware, CoCoNuT’s median execution time to validate a bug is six sec on Codeflaws and 6 min on Defects4J (benchmark with the largest programs and test suites).

CoCoNuT median **end-to-end time** to fix a bug varies from 16.7 min on Codeflaws to 22.7 min on Defects4J. In comparison, on identical hardware, the median time of other tools that we ran on Codeflaws (Angelix, GenProg, SPR, and Prophet) varies from 30 sec to 4 min. TBar, on the same hardware, has an 8 min median execution time on Defects4J.

While the end-to-end approach of CoCoNuT is slower than existing approaches, it is still reasonable. In addition, we can shorten the execution time by reducing the number of patches generated in inference. Generating only 1,000 patches (i.e., 50 patches per model) would reduce CoCoNuT’s end-to-end time to 2 min on Codeflaws and 7 min on Defects4J while still fixing most of the bugs (e.g., 34 in Defects4J). Parallelism on multiple GPUs and CPUs can also speed up the inference.

## 6 LIMITATIONS

The dataset used to train our approach is different from the ones used by other work (e.g., SequenceR and DLFix), which could impact our comparison results. However, the choice of datasets and how to extract and represent data are a key component of a technique. In addition, we compare our approach with LSTM, FConv, and Transformer architectures using the same training data and hardware, which shows that CoCoNuT outperforms all other three. Finally, both DLFix and SequenceR use data that was unavailable at the time of the bug fix (i.e., their models are trained using instances committed after the bugs in Defects4J were fixed), which could produce false good results as shown by prior work [75].

A challenge of deep learning is to explain the output of a neural network. Fortunately, for developers, the repaired program that compiles and passes test cases should be self-explanatory. For users who build and improve CoCoNuT models, we leverage the recent multi-step attention mechanism [20] to explain why a fix was generated or not.

There is a threat that our approach might not be generalizable to fixing bugs outside of the tested benchmarks. We use six different

benchmarks in four different programming languages to address this issue. In the future, it is possible to evaluate our approach in additional benchmarks [24, 57, 70].

There is randomness in the training process of deep-learning models. We perform multiple runs and find that the randomness in training has little impact on the performances of the trained models.

## 7 RELATED WORK

**Deep Learning for APR:** SequenceR [13], DLFix [42], and Tufano et al. [77] are the closest work related to CoCoNuT. The main differences with CoCoNuT are that these approaches use RNN (a single LSTM-based NMT model for SequenceR and Tufano et al., and a TreeRNN architecture for DLFix) and represent both the buggy line and its context as one input. These approaches have trouble extracting long term relations between tokens and do not capture the diversity of bug fixes. We showed in Section 5.3 that CoCoNuT outperforms both DLFix and SequenceR. Tufano et al. [77] generates templates instead of complete patches and thus cannot be directly compared. Deep learning has also been used to detect and repair small syntax [73] and compilation [23, 60] issues (e.g., missing parenthesis). These models show promising results for fixing compilation issues but only learn the syntax of the programming language.

**G&V Program Repair:** Many APR techniques have been proposed [8, 12, 27, 29, 37, 38, 50, 52, 54, 58, 64, 69, 82, 86–88]. We use a different approach compared to these techniques, and as shown in Section 5, our approach fixes bugs that existing techniques have not fixed. In addition, these techniques require significant domain knowledge and manually crafted rules that are language-dependent, while thanks to our context-aware ensemble NMT approach, CoCoNuT automatically learns such patterns and is generalizable to several programming languages with minimal effort.

**Grammatical Error Correction (GEC):** Recent work uses machine translation to fix grammar errors [14, 15, 18, 19, 30, 32, 51, 63, 72, 74, 90]. Among them, [15] applied an attention-based convolutional encoder-decoder model to correct sentence-level grammatical errors. CoCoNuT is a new application of NMT models on source code and programming languages, addressing unique challenges. Studying whether our new context-aware NMT architecture improves GEC remains future work.

**Deep Learning in Software Engineering:** The software engineering community had applied deep learning to perform various tasks such as defects prediction [40, 79, 81], source code representation [6, 7, 66, 80], source code summarization [4, 22] source code modeling [5, 11, 26, 83], code clone detection [41, 84], and program synthesis [3, 44, 61, 62]. Our work uses a new deep learning approach for APR.

## 8 CONCLUSION

We propose CoCoNuT, a new end-to-end approach using NMT and ensemble learning to automatically repair bugs in multiple languages. We evaluate CoCoNuT on six benchmarks in four different programming languages and find that CoCoNuT can repair 509 bugs including 309 that have not been fixed before by existing

techniques. In the future, we plan to improve our approach to work on multi-hunk bugs.

## ACKNOWLEDGMENT

The authors thank Shruti Dembla for her contribution in collecting java projects from GitHub. The research is partially supported by Natural Sciences and Engineering Research Council of Canada, a Facebook research award, and an NVIDIA GPU grant.

## REFERENCES

- [1] Fairseq-py. <https://github.com/pytorch/fairseq>, 2018.
- [2] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 2019.
- [3] Carol V Alexandru. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1068–1070. ACM, 2016.
- [4] Miltiadis Allamanis, Haifeng Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.
- [5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.
- [6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=BJOFETxR->.
- [7] Uri Alon, Meital Zilberman, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POLY):40, 2019.
- [8] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSM)*, pages 328–332. IEEE, 2019.
- [9] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
- [10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.
- [11] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2tree neural translation model for learning source code changes. *arXiv preprint arXiv:1810.00314*, 2018.
- [12] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 637–647. IEEE, 2017.
- [13] Zimin Chen, Steve James Kommarusich, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.
- [14] Shamil Chollampatt and Hwee Tou Ng. A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction. 2018. URL <http://arxiv.org/abs/1801.08831>.
- [15] Shamil Chollampatt and Hwee Tou Ng. A multilayer convolutional encoder-decoder neural network for grammatical error correction. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, February 2018.
- [16] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 933–941. JMLR.org, 2017.
- [17] Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, pages 85–91. ACM, 2016.
- [18] Tao Ge, Furu Wei, and Ming Zhou. Reaching Human-level Performance in Automatic Grammatical Error Correction: An Empirical Study. (3):1–15, 2018. URL <http://arxiv.org/abs/1807.01270>.
- [19] Tao Ge, Furu Wei, and Ming Zhou. Fluency Boost Learning and Inference for Neural Grammatical Error Correction. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 1:1–11, 2018. URL <http://aclweb.org/anthology/P18-1097>.
- [20] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning, pages 1243–1252, 2017.
- [21] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21, 2012.

- [22] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.
- [23] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.
- [24] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugjs: A benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.
- [25] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.
- [26] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, pages 763–773, 2017.
- [27] Jinru Hu, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 888–891. ACM, 2018.
- [28] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker. Perplexity – a measure of the difficulty of speech recognition tasks. *Journal of the Acoustical Society of America*, 62:S63, November 1977. Supplement 1.
- [29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. pages 298–309, 2018.
- [30] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Shubha Guha, and Kenneth Heafield. Approaching Neural Grammatical Error Correction as a Low-Resource Machine Translation Task. (2016):595–606, 2018. URL <http://arxiv.org/abs/1804.05940>.
- [31] René Just, Dariush Jalali, and Michael D Ernst. Defectsj4: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.
- [32] Masahiro Kaneko, Yuya Sakaizawa, and Mamoru Komachi. Grammatical Error Detection Using Error- and Grammaticality-Specific Word Embeddings. *Proceedings of the 8th International Joint Conference on Natural Language Processing*, (2016):40–48, 2017. URL <https://github.com/kanekomasahiro/grammatical-error->.
- [33] Jindao Kim and Sunghun Kim. Automatic patch generation with context-based change application. *Empirical Software Engineering*, 24(6):4071–4106, 2019.
- [34] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 481–490. IEEE, 2011.
- [35] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791*, 2018.
- [36] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325, 2019.
- [37] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. 1:213–224, 2016.
- [38] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.
- [39] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.
- [40] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. Software defect prediction via convolutional neural network. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 318–328. IEEE, 2017.
- [41] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.
- [42] Yi Li, Wang Shaohua, and Tien N. Nguyen. DLfix: Context-based code transformation learning for automated program repair. In *Software Engineering (ICSE), 2020 IEEE/ACM 42nd International Conference on*. IEEE, 2020.
- [43] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.
- [44] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.
- [45] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662. IEEE, 2018.
- [46] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 102–113. IEEE, 2019.
- [47] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, pages 31–42, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330577. URL <http://doi.acm.org/10.1145/3293882.3330577>.
- [48] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–12. IEEE, 2019.
- [49] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé François D Assise Bissyande, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020.
- [50] Xuliang Liu and Hao Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129. IEEE, 2018.
- [51] Zhuo Ran Liu and Yang Liu. Exploiting Unlabeled Data for Neural Grammatical Error Detection. *Journal of Computer Science and Technology*, 32(4):758–767, 2017. ISSN 18604749. doi: 10.1007/s11390-017-1757-4.
- [52] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.
- [53] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 702–713. IEEE, 2016.
- [54] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, volume 2017, 2017.
- [55] Fan Long et al. Automatic patch generation via learning from successful human patches. PhD thesis, Massachusetts Institute of Technology, 2018.
- [56] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.
- [57] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.
- [58] Matias Martinez and Martin Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.
- [59] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.
- [60] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936, 2019.
- [61] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211*, 2015.
- [62] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=HkfXMz-Ab>.
- [63] Courtney Napoles and Chris Callison-Burch. Systematically Adapting Machine Translation for Grammatical Error Correction. *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 345–356, 2017. URL <http://www.aclweb.org/anthology/W17-5039>.
- [64] Frolin S Ocariza, Jr, Karthik Pattabiraman, and Ali Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the 36th International Conference on Software Engineering*, pages 837–847, 2014.
- [65] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.
- [66] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

- [67] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. Does genetic programming work well on automated program repair? In *2013 International Conference on Computational and Information Sciences*, pages 1875–1878. IEEE, 2013.
- [68] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.
- [69] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. Elixir: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. IEEE Press, 2017.
- [70] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 10–13, 2018.
- [71] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, pages 13–24. IEEE Press, 2019.
- [72] Keisuke Sakaguchi, Matt Post, and Benjamin Van Durme. Grammatical Error Correction with Neural Reinforcement Learning. 2017. URL <http://arxiv.org/abs/1707.00299>.
- [73] Eddie A Santos, Joshua C Campbell, Abram Hindle, and José Nelson Amaral. Finding and correcting syntax errors using recurrent neural networks. *PeerJ PrePrints*, 2017.
- [74] Allen Schmaltz, Yoon Kim, Alexander M. Rush, and Stuart M. Shieber. Adapting Sequence Models for Sentence Correction. 2017. URL <http://arxiv.org/abs/1707.09067>.
- [75] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 99–108. IEEE, 2015.
- [76] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Code-flaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.
- [77] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, pages 832–837, 2018.
- [78] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.
- [79] Jinyong Wang and Ce Zhang. Software reliability prediction using a deep learning model based on the rnn encoder–decoder. *Reliability Engineering & System Safety*, 2017.
- [80] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163*, 2017.
- [81] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 297–308. IEEE, 2016.
- [82] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1–11. ACM, 2018.
- [83] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.
- [84] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM, 2016.
- [85] Tong Xiao, Tian Xie, Yi Yang, Chang Huang, and Xiaogang Wang. Learning from massive noisy labeled data for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2691–2699, 2015.
- [86] Qi Xin and Steven P Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670. IEEE Press, 2017.
- [87] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.
- [88] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lameiras Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.
- [89] Jingqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.
- [90] Helen Yannakoudakis, Marek Rei, Øistein E Andersen, and Zheng Yuan. Neural Sequence-Labelling Models for Grammatical Error Correction. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2795–2806, 2017. doi: 10.18653/v1/D17-1297. URL <http://aclweb.org/anthology/D17-1297>.
- [91] Yuan Yuan and Wolfgang Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.

# Detecting and Diagnosing Energy Issues for Mobile Applications

Xueliang Li

College of Computer Science and  
Software Engineering  
Shenzhen University  
Shenzhen, China

Yuming Yang

College of Computer Science and  
Software Engineering  
Shenzhen University  
Shenzhen, China

Yepang Liu

Department of Computer Science  
and Engineering  
Southern University of Science  
and Technology  
Shenzhen, China

John P. Gallagher

Department of People and  
Technology  
Roskilde University  
Roskilde, Denmark  
IMDEA Software Institute  
Madrid, Spain

Kaishun Wu

College of Computer Science and  
Software Engineering  
Shenzhen University  
Shenzhen, China

## ABSTRACT

Energy efficiency is an important criterion to judge the quality of mobile apps, but one third of our randomly sampled apps suffer from energy issues that can quickly drain battery power. To understand these issues, we conducted an empirical study on 27 well-maintained apps such as Chrome and Firefox, whose issue tracking systems are publicly accessible. Our study revealed that the main root causes of energy issues include unnecessary workload and excessively frequent operations. Surprisingly, these issues are beyond the application of present technology on energy issue detection. We also found that 25.0% of energy issues can only manifest themselves under specific contexts such as poor network performance, but such contexts are again neglected by present technology.

In this paper, we propose a novel testing framework for detecting energy issues in real-world mobile apps. Our framework examines apps with well-designed input sequences and runtime contexts. To identify the root causes mentioned above, we employed a machine learning algorithm to cluster the workloads and further evaluate their necessity. For the issues concealed by the specific contexts, we carefully set up several execution contexts to catch them. More importantly, we designed leading edge technology, e.g. pre-designing input sequences with potential energy overuse and tuning tests on-the-fly, to achieve high efficacy in detecting energy issues. A large-scale evaluation shows that 91.6% issues detected in our experiments were previously unknown to developers. On average, these issues double the energy costs of the apps. Our testing technique achieves a low number of false positives.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397350>

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Mobile Applications, Energy Issues, Energy Bugs, Android

### ACM Reference Format:

Xueliang Li, Yuming Yang, Yepang Liu, John P. Gallagher, and Kaishun Wu. 2020. Detecting and Diagnosing Energy Issues for Mobile Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397350>

## 1 INTRODUCTION

Mobile devices have evolved into a wide ecosystem, providing millions of third-party apps to serve various user needs. Energy efficiency is a desirable quality attribute of mobile apps. However, many real-world mobile apps suffer from energy misuses. For example, Huang et al. [32] reported that most energy issues in mobile devices are caused by apps (47.9%) rather than systems (22.2%). We also randomly sampled 89 open-source Android apps and found that 27 (30.3%) of them suffer from serious software energy issues.

Despite many pieces of existing work (e.g., [2, 31, 32, 39]), the root causes and manifestations of energy issues in mobile apps are still not very well studied. Due to this reason, there exist few effective testing techniques to uncover serious energy issues. This motivates us to conduct an empirical study on 27 energy-inefficient Android apps to learn the real root causes of energy issues and their manifestation in practice. Specifically, we aim to answer two research questions:

- **RQ1 (Issue Causes):** *What are the common root causes of energy issues?*
- **RQ2 (Issue Manifestation):** *How do energy issues manifest themselves in practice?*

We analyzed 171 energy issues from the 27 open-source projects. For RQ1, we identified four main root causes of energy issues: 1) unnecessary workload, 2) excessively frequent operations, 3) wasted

**Table 1: Comparison with the existing technology in terms of ability to deal with root causes and manifestations.**

Root Cause	Existing work [2]	This work
Unnecessary Workload ( 46.7%)	✗	✓
Excessively Frequent Operations ( 17.8%)	✗	✓
Wasted Background Processing ( 20.0%)	✓	✓
No Sleep ( 32.2%)	✓	✓
Manifestation Type	Existing work [2]	This work
Simple Inputs ( 9.2%)	✓	✓
Special Inputs ( 68.4%)	✓	✓
Special Context ( 25.0%)	✗	✓

*background processing*, and 3) *no-sleep* (**Finding 1**). For RQ2, we found that many energy issues require special inputs (68.4%) or special context (25.0%) to trigger and only 9.2% energy issues can manifest themselves with simple inputs (**Finding 2**).

We further studied the state-of-the-art testing technique for energy issues [2], which was proposed by Banerjee et al., and made two observations. First, as shown in Table 1, the existing technique is only capable of exposing issues resulting from wasted background processing and no sleep. This is because the technique diagnoses energy issues based on *E/U ratio*, i.e., the ratio of energy-consumption to hardware-utilization. If E/U ratio is high, it means that energy consumption is high, while hardware utilization is low, implying that the app under analysis is energy-inefficient. This point of view seems reasonable. For energy issues caused by wasted background processing and no-sleep, the E/U ratio could be remarkably high. However, according to Finding 1, many energy issues can also be caused by unnecessary workload and excessively frequent operations. These issues cannot be detected via analyzing E/U ratio: they may enlarge E and U simultaneously, hence E/U ratio is not a good indicator of the existence of such issues. Second, regarding Finding 2, the existing technique is capable of generating simple and special inputs to trigger energy issues, but does not simulate special contexts (e.g., poor network performance). Due to this limitation, it may miss many real energy issues (25.0% of our studied issues can only be triggered under special context).

Based on these observations, we propose a novel testing framework for effectively detecting energy issues. It works in two critical steps. First, our framework examines apps with a large variety of well-designed input sequences and runtime contexts, aiming to provoke energy issues to manifest themselves. Next, we need to recognise the appearance of energy issues. This is challenging due to the lack of effective test oracle. To address the challenge, our framework deals with different issue causes respectively. For wasted background processing, we compute statistical dissimilarity of the power traces before and after use of the app. If the dissimilarity exceeds a threshold, it reveals that the backgrounded (i.e. after-use) app is not as moderate as supposed to be, and is probably suffering from energy issues. We handle no-sleep in an analogue manner. To detect unnecessary workload and excessively frequent operations, we employ a machine learning algorithm to assess the *necessity* of the app's workloads according to several key criteria, such as lengths of continuous-high-power periods. If the workload is assessed *unnecessary* and *severely* energy consuming, then it will be identified as the occurrence of an energy issue.

We also design a package of practical techniques to enhance issue-detection efficacy of framework. For instance, before testing, our framework scans and analyses the source code of apps to extract the input sequences that are most likely to incur energy issues. During testing, the framework adjusts test targets on-the-fly to increase chances of exposing energy issues.

We performed experiments to evaluate our framework. The results show that our testing framework can uncover a large number of serious energy issues in high-quality apps, 91.6% of which have never been discovered before. On average, these issues double the energy cost of the apps. Manual verification also shows that our framework only reports a low number of false positives.

The key contributions of this paper are as followed:

- To the best of our knowledge, we conducted the largest-scale empirical study on developer-reported energy issues in mobile apps (*largest previous empirical study* [30]: 8 app subjects, 10 energy issues; *this paper*: 27 app subjects, 171 energy issues). Our findings can greatly benefit the research on energy issue detection and diagnosis.
- Inspired by the findings, we designed and implemented an automated testing framework for detecting energy issues. Our innovations include extracting battery-hungry input sequences from source code, steering the test direction on-the-fly for high detection efficacy, and employing machine learning to classify app workload.
- We empirically evaluated our framework and the results are promising: it detected 83 issues in 89 apps, creating many opportunities for optimizing the energy efficiency of these apps. As far as we know, this evaluation of a testing framework for energy issue detection is of the largest scale (*largest previous evaluation* [2]: 30 app subjects and detected 12 issues; *this paper*: 89 app subjects and detected 83 issues). Our subjects are also of higher quality than previous work, which are selected considering metrics such as high popularity and maintenance quality. In contrast, most subjects in previous work do not meet this standard.

In the remainder of this paper, we first introduce the data source for empirical study in Section 2, and discuss the findings in Section 3. We present our testing framework and technical details in Sections 4 and 5. Finally, we present an evaluation of our framework and discuss the results in Section 6.

## 2 DATA SOURCE

Open-source projects typically have publicly accessible issue tracking systems and code repositories. In the issue tracking systems, developers can post an issue report, which contains a title and a main body part, to report the symptoms of their observed bug/issue<sup>1</sup> and the steps to reproduce the issue (optional). Following that, developers can discuss the issue and comment on the report. Those developers who are assigned to fix the issue can propose potential code revisions. Typically, after code review by other project members and further changes, such revisions will be committed to the project's code repository.

<sup>1</sup>We may use the terms of bugs and issues interchangeably in this paper.

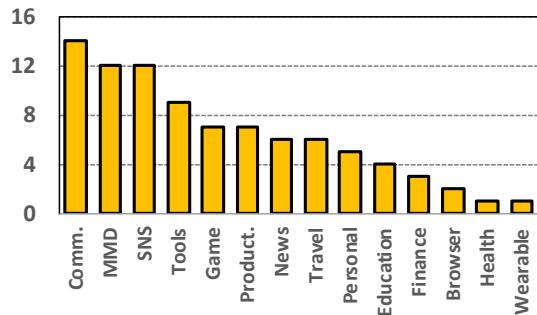


Figure 1: The 89 app subjects of different categories.

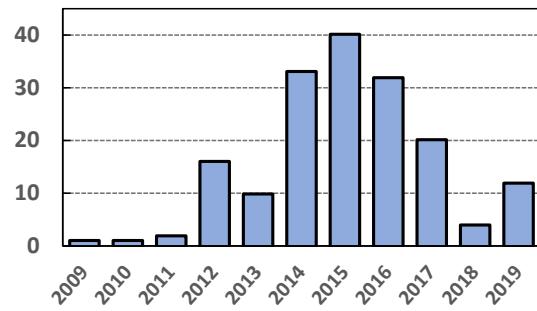


Figure 2: The yearly number of reported energy issues.

Our empirical study is conducted on well-maintained Android application projects from three popular open-source software hosting platforms: GitHub<sup>2</sup>, Mozilla<sup>3</sup>, and Chromium<sup>4</sup> repositories. The criteria for selecting app subjects for our study are these: 1) a subject should have achieved at least 1,000 downloads on the market (popularity), 2) it should have more than one hundred code revisions (maintainability). Following these criteria, we randomly selected 89 app subjects from those three software hosting platforms. These open-source applications are also indexed by the F-Droid database<sup>5</sup>. Figure 1 presents the numbers of app subjects of different categories. They cover most application categories in F-Droid and in total contain 108,193 issue reports, indicating that these subjects are quite large-scale.

To search for energy issues, we employed keyword searching in issue reports' title and body to locate potential energy issues. The keywords we used are *energy*, *power* and *battery*. While keyword searching generally helps to retrieve most energy-related issue reports, it can also produce false positive results when the issue reports accidentally contain any of our keywords. To filter out such irrelevant issue reports, we manually verified each returned issue report to make sure the issue concerned is indeed an energy issue. In total, we checked 976 retrieved issue reports and this helped us locate 171 real energy issue reports from 27 apps. Figure 2 shows the distribution of opening dates of these reports. There was a peak reporting period from 2014 to 2017. After, the number dropped in 2018 and 2019. Only 28.1% (48 out of 171) reported issues were fixed. The main reasons for no fixes are these: 1) many energy issues are irreproducible (60.8%), 2) the problematic code cannot

be localized (9.3%), 3) energy-saving by fixing the issues cannot be evaluated (9.3%), and 4) the fix causes other issues (9.3%). So in this paper, we will present a cutting-edge technology for effectively pinpointing energy issues in the lab where testing condition is precisely controlled, so the issues can be strictly reproduced and localized. Also, energy-saving can be accurately evaluated.

### 3 EMPIRICAL STUDY

To answer our research questions, we carefully studied the 171 energy issue reports. The results are as follows.

#### 3.1 RQ1: What Are the Common Root Causes of Energy Issues?

Among the 171 reports, 90 explicitly show the information on root causes of the issues. We examined all of them and observed the following six root causes. Some issues were caused by multiple reasons, hence, the sum of the percentages below is over 100%.

**Unnecessary workload** (42/90=46.7%). Many applications perform certain computations that do not deliver perceptible benefits to users. These computations incur unnecessary workload on hardware components including CPU, GPU, GPS, network interface, and screen display. For example, in the reports of Chrome issue 662012 and 541612<sup>6</sup>, the application produces frames constantly even when visually nothing is changed or repainted, which causes huge workload on CPU and GPU. And the report of OpenGPSTracker issue 406 shows that the app keeps recording users' location even after not moving for minutes, barely exhausting battery.

**Excessively frequent operations** (16/90=17.8%). Performing certain operations too frequently can also waste power. In comparison with unnecessary workload, when fixing energy inefficiencies caused by excessively frequent operations, the developers do not completely remove the operations (because their functionality is necessary), but reduce the frequency of operations. For example, in Firefox (issue 979121), whenever users type in the URL bar and the text changes, the app will query the database (e.g. for auto-completion). Considering users often visit the websites they visited before, developers suggested to store users' browsing history in memory to reduce database hits to save energy.

**Wasted background processing** (18/90=20.0%). As battery powered mobile devices are extremely sensitive to energy dissipation, it is good practice to make backgrounded applications as quiet as possible. Specifically, the "background" here means that after the use of an application or "activity" (a major type of application component that represents a single screen with a user interface)<sup>7</sup>, users press the *Home* button or switch to another application or activity, so the previous application or activity goes to background. Typical examples are Chrome issue 781686 and Firefox issue 1022569: when users select a new tab (each tab is an activity), the invisible old tab would still keep being reloaded, which wastes battery. But these issues do not have an easy solution since users may want old tabs to be reloaded.

<sup>2</sup><https://github.com>

<sup>3</sup><https://dxr.mozilla.org>

<sup>4</sup><https://www.chromium.org>

<sup>5</sup><https://f-droid.org/>

<sup>6</sup>"Chrome" is the app name, "662012" and "541612" are the issues' ID given by the corresponding issue tracking system.

<sup>7</sup><https://developer.android.com/guide/components/fundamentals.html>

**No-sleep** (29/90=32.2%). The no-sleep issue means that when the screen is off and device is supposed to enter sleep mode, certain apps still keep the device awake, which usually results from misuse of asynchronous mechanisms [35] like services, broadcast receivers, alarms and wake-locks. For example, Kontalk issue 143 unnecessarily holds a wake-lock, preventing the device from falling asleep. For another example, in Firefox (issue 1026669), the Simple Service Discovery Protocol (SSDP) activates the searching service every two minutes when the screen is off, which not only incurs a large amount of workload but also prevents the device from entering the sleep mode. Program 1 gives the JavaScript patch for fixing this issue. It added the cases to deal with “application-background” and “application-foreground” for the SSDP service. Note that, the “application-background” defined by developers includes both screen-off time and the scenarios where users switch to another application. So this issue belongs to two categories: *no-sleep* and *wasted background processing*.

---

```

case "ssdp-service-found":
- {
- this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
- break;
- }
+ this.serviceAdded(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
case "ssdp-service-lost":
- {
- this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
- break;
- }
+ this.serviceLost(SimpleServiceDiscovery.findServiceForID(aData));
+ break;
+ case "application-background":
+ // Turn off polling while in the background
+ this._interval = SimpleServiceDiscovery.search(0);
+ SimpleServiceDiscovery.stopSearch();
+ break;
+ case "application-foreground":
+ // Turn polling on when app comes back to foreground
+ SimpleServiceDiscovery.search(this._interval);
+ break;

```

---

**Program 1: JavaScript patch of Firefox issue 1026669.**

**Spike workload** (2/90=2.2%). A spike workload can cause lagging UI [30], degrade user experience and heat up the device, inducing a huge energy waste. For instance, in RocketChat (issue 3321), when users send or receive GIF animation pictures, CPU utilization quickly rises to 100% and heavily affects the battery.

**Runtime exception** (3/90=3.3%). In some cases, runtime exceptions may provoke abnormal behaviors of a mobile application and cause energy waste. For instance, in AntennaPod (issue 1796), the “NullPointerException” makes the download process persist and consume power. In our study, such energy issues caused by runtime exceptions are not common and we only observed three cases.

*Considering spike workload and runtime exception are of small proportions in practice, our issue-detection technology only focuses on the main root causes apart from them.*

### 3.2 RQ2: How Do Energy Issues Manifest Themselves in Practice?

Out of the 171 reports, 76 contain explicit information that shows how the issues manifest themselves. We studied these 76 issues to

answer RQ2. We observed three manifestation types: *simple inputs*, *special inputs* and *special context*. It is worthwhile to notice that, *simple inputs* and *special context* may combine to incur issues, on the other hand, *special inputs* and *special context* may also overlap.

**Simple inputs** (7/76=9.2%). Simple inputs mean one tap or swipe gesture in common interaction scenarios. We found seven issues are of this type of manifestation. For example, Andlytics issue 543 lets the app refresh itself whenever the user opens the app. And VoiceAudioBookPlayer issue 299 makes the app unnecessarily scan folders every time the user starts or leaves the app.

**Special inputs** (52/76=68.4%). The majority of the energy issues can only be triggered with certain specific inputs or a sequence of user interactions (e.g. text typing, taps, or swipes) under certain states of an application. For instance, the c:geo (a geocaching app) issue 4704 requires three steps to reproduce: 1) open the app and make sure GPS is inactive since the app starts, 2) change between cache details and other tabs of the same geocache, so GPS is activated, 3) put the device in standby and let timeout to screen-off. After a while, users would find GPS stays active even when the screen is turned off. To avoid energy waste, users decide to quit using the app.

**Special Context** (19/76=25.0%). Special context includes environmental conditions (rather than user interactions, e.g. taps) such as the accessibility of networks, location of the device, settings of the OS and applications. In our dataset, 19 issues require such special contexts to trigger. For instance, MPDroid issue 3 appears when the user is watching stream videos but the network is disconnected; the app then keeps trying to load the video and consumes battery. AnkiDroid issue 2768 occurs when users lock the phone screen when the application is in “review” mode and a notification comes in afterwards, so the screen will hold on until the battery is dead.

## 4 OVERVIEW OF TESTING FRAMEWORK

The following observations motivated us to design a novel testing framework for effectively detecting energy issues in real apps:

- From **Finding 1**, we address previously unaddressed energy issues caused by unnecessary workload and excessively frequent operations.
- From **Finding 2**, we found that 25.0% of energy issues can only be manifested under special context such as poor network performance. However, such factors were neglected.

According to the first observation, as we discussed in Section 1, *evaluating the necessity of app workload is crucial for identifying these issues*. We will use machine learning to cluster workloads, and further assess their necessity, as shown later in Section 5.3. According to the second observation, we will devise two types of most common special contexts for effectively revealing these issues, as shown in Section 5.1.2. Developers also can duplicate our experiment for detecting these hidden issues. Importantly, we also make practical designs and implementations to enhance the efficacy of our testing framework.

Figure 3 shows the framework overview. The framework first makes sophisticated preparation before testing. For example, it inspects the source code and collects the candidate input-sequences that are most suspected of energy overuse. A set of candidate runtime contexts (containing the above mentioned two types of special

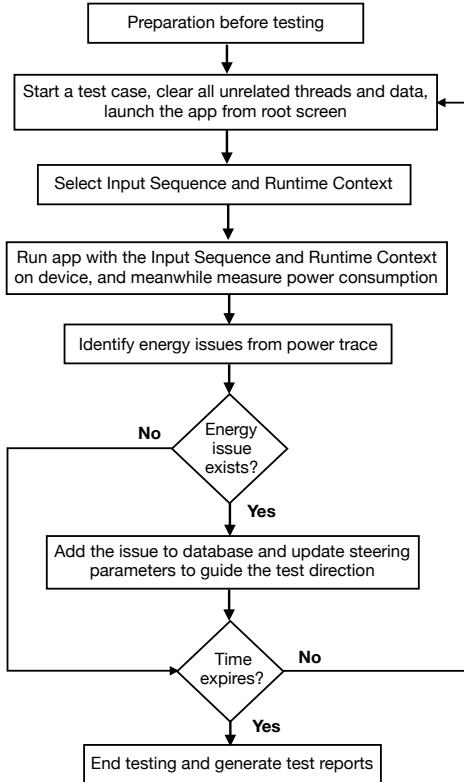


Figure 3: The flow chart for our testing framework.

contexts) are also carefully designed to increase the chance of provoking energy issues. Later, these candidate inputs and contexts will be explored under an effective and systematic scheme.

To start a test case, the framework at first clears unrelated threads and data to minimize the interference from other applications and previous test cases. Then, it will select one input sequence and one runtime context from the candidates, then run the app with them. During the entire test, the power consumption of device is traced with a power monitor. Our framework will look into the power trace and decide whether an energy issue exists. If one does exist, the issue information will be added into database. The entire test is limited with a time budget. If the time budget runs out, the test will quit and test reports will be generated for developers to help fix the issues. Otherwise, our framework will start a new test case.

As mentioned above, our framework explores the inputs and contexts under a systematic scheme, where the exploring direction is tuned on-the-fly. The rationale of our scheme is that if an energy issue occurs, it implies that the type of the input sequence and runtime context incurring this issue may be more likely to uncover energy issues than average case, since it did cause an energy issue to show up; we thus increase the chance of this type of inputs and context to be tested. Concretely, we utilize a set of parameters and iteratively update them to guide the test direction, as shown later in Section 5.2.

The large-scale evaluation (Section 6) shows that, exploiting these practical and targeted tests, our framework largely outperforms the state of the art on the efficacy in detecting all kinds of energy issues.

## 5 DETAILED TECHNOLOGY

This section introduces detailed implementations of our testing framework. It involves how to design candidate input sequences and runtime contexts, how to steer the test direction at runtime, and how to identify energy issues from the power traces, etc. The overall objective is to effectively and accurately pinpoint energy issues.

### 5.1 Preparation before Testing

As shown in Section 4, our test case is driven by the input sequence and runtime context. Our candidate input sequences are designed for high utilization of the main hardware components, such as CPU, screen display and network interface, since they are usually the culprits of energy waste, as shown in the literature [46]. Meanwhile, according to Finding 2, we will carefully devise a set of artificial runtime contexts for effectively detecting those energy issues.

**5.1.1 Design of Candidate Input Sequences.** We design two types of candidate input sequences. One is **weighted input sequences**, the other is **random input sequences**. The former helps our framework detect issues in a guided manner, the latter will cover some corner cases that are hard to predict.

**Weighted input sequences** are generated referring to the Event-Flow Graph (EFG) [36]. Each node in an EFG is a User Interface (UI) component, such as a button or a list item. If a user interaction on a UI component, say  $node_1$ , can immediately bring out another UI component, say  $node_2$ , then the EFG has a directed edge from  $node_1$  to  $node_2$ . Technically, we utilize Layout Inspector<sup>8</sup> to construct the EFG of an app. An arbitrary path in EFG could be a candidate input sequence for our testing framework. In practice, our test cases always start from the root node (i.e. the initial UI component of the app). The lengths of paths are constrained with a limit. Note that, even though Layout Inspector can construct the EFG, it does not have a capacity to run apps with the paths. So in our test, we use Dynodroid<sup>9</sup> to feed the paths to apps.

Importantly, all input sequences generated from EFG are assigned a weight. The weight indicates potential of a sequence to cause energy waste; an input sequence with a larger weight has a higher priority to be tested. We use Equation (1) to calculate the weight for each input sequence.  $S$  is the number of a certain set of system APIs invoked by the input sequence.  $C$  is the number of function invocations and block transitions incurred by the input sequence.  $\alpha$  and  $\beta$  are used for adjusting influences of  $S$  and  $C$  on the weight;  $\alpha > 0, \beta > 0, \alpha + \beta = 1$ . In practice, we set  $\alpha$  at 0.6 and  $\beta$  at 0.4.

$$weight = \alpha * S + \beta * C \quad (1)$$

The reason why we resort to  $S$  and  $C$  to indicate the potential of excessive energy use is this: as shown in [46], the main energy-consuming components are CPU, screen display, network interface (cellular and WiFi), as well as GPS and various sensors. Except for CPU, all other components can only be controlled by a set of system APIs, as listed in [2]. The more this set of system APIs an input sequence accesses, the larger are the chances it causes energy waste. The CPU executes basic operations that constitute the source

<sup>8</sup><https://developer.android.com/studio/debug/layout-inspector>

<sup>9</sup><https://dynodroid.github.io>

code of apps, for example, arithmetic operations like additions and multiplications, and control-flow operations mainly including function invocations and block transitions. Recent research [22, 26] has shown that control-flow operations are the actual main energy-consumers for Android app source code. We therefore use the total number of the main control-flow operations (i.e. function invocations and block transitions) to indicate the potential CPU overload of an input sequence.

Note again that, we calculate  $S$  and  $C$  before testing. We first instrument the app source code, and run the app with the input sequences in the emulator on a powerful PC, and then record their  $S$  and  $C$  values individually.

Apart from **weighted input sequences**, we also designed **random input sequences** to cover exceptional cases we might not envisage. We use the Monkey tool<sup>10</sup> to generate random input sequences, such as taps and swipes. “Random” here means the position of the inputs on screen are randomly set. Monkey does not generate input sequences at runtime. Instead, Monkey pre-defined a large number of random input sequences. When being asked for an input sequence, Monkey will arbitrarily deliver one of them with its ID. The advantage of this design is that testers can conveniently use the same ID to repeat the input sequence and reproduce the test case. In our test, the pre-defined input sequences in Monkey are all adopted as our candidate input sequences.

In summary, our test combines two types of input sequences, namely, **weighted** and **random**. In Section 5.2, we will show the strategy for balancing the testing time on them.

**5.1.2 Design of Candidate Runtime Contexts.** The entire experiment is set in a signal shielding room, enabling us to manipulate the contextual factors, such as the strength of WiFi (in our test, we employ WiFi as the connection to Internet) and GPS signal. We designed three types of runtime contexts, namely, Normal, Network Fail and Flight Mode. In Normal, the WiFi and GPS work normally (package delivery delay is 36 ms and bandwidth is 3.2 Mb/s). In Network Fail, the signal is seriously weak (package or message delivery delay lengthens to 451 ms, bandwidth drops to 12.0 Kb/s). In Flight Mode, WiFi is closed at software level by the OS, and GPS works normally.

The **reason** for choosing Network Fail and Flight Mode as representatives for special contexts is that our empirical study shows they are the two major types of special contexts. The former accounts for 26.3% (5 out of 19), the latter for 15.8% (3 out of 19) of issues manifested under special context.

We also designed a special type of runtime context, Non-background. We designed this context because in our experiment we observed that it can provoke more no-sleep issues, as shown later in Section 6.3. In Non-background, the network and GPS work ordinarily, however, we do not input a press of *Home* button to the device after EXECUTION stage (the stage-division for test cases will soon be explained in Section 5.3). That is, the test case does not have BACKGROUND stage, and straight goes to SCREEN-OFF.

<sup>10</sup><https://developer.android.com/studio/test/monkey.html>.

## 5.2 Steer the Test Direction On-the-fly

Our framework steers test direction dynamically based on test history. Algorithm 1 shows details of our steering scheme. The rationale behind it is this: when an energy issue is detected, it implies that this type of input sequence and runtime context may have a greater opportunity than usual to provoke energy issues since it did trigger an energy issue. Hence, our framework will generate slightly more of this type of test cases for larger chance of confronting energy issues.

**Algorithm 1:** Steer the test direction on-the-fly

```

Data:
WeightedInputSequences = {(sequencei, weighti)};
RandomInputSequences = {sequencej};
RuntimeContexts[N] = {contextk};
0 < pwgt < 1, Pctx[N] = {0 < pk < 1};
Δwgt, Δctx;
1 Preparation before testing;
2 Start a test case, clear unrelated threads and data, launch app from root screen;
3 #-----Select Input Sequence and Runtime Context-----#
4 Determine the type of input sequence, and the type of “weighted” has a
   probability of pwgt to be chosen;
5 if the determined type is “weighted” then
6   | Select an unexplored sequence with highest weight in
     WeightedInputSequences;
7 else
8   | Randomly select one sequence from RandomInputSequences;
9 end
10 Select one context from RuntimeContexts with its corresponding
   probability;
11 #-----#
12 Run app with the selected input sequence and runtime context on device, and
   meanwhile measure power consumption;
13 Identify energy issues from power trace;
14 if there exists an energy issue then
15   | Add the energy issue to database;
16   #-----Update steering parameters-----#
17   if the issue is incurred by a “weighted” sequence then
18     | if pwgt ≤ wgt_up_threshold - Δwgt then
       |   Pwgt := pwgt + Δwgt;
19   else
20     | if pwgt ≥ wgt_down_threshold + Δwgt then
       |   Pwgt := pwgt - Δwgt;
21   end
22 switch which context triggers the energy issue do
23   case e.g. contextk do
24     | if pk ≤ cxt_up_threshold - Δcxt and there are n elements
       |   (except pk) in Pctx that are greater than or equal to
       |   cxt_down_threshold + Δcxt and n > 0 then
25       |   | pk := pk + Δcxt;
26       |   | Decrease those n elements individually by Δcxt/n;
27     end
28   end
29 end
30 #-----#
31 end
32 if time expires then End Testing and generate test reports;
33 Go back to line 2

```

**Data for the algorithm.** The candidate input sequences and runtime contexts are designed based on the approach we demonstrated in Section 5.1.1 and 5.1.2. We present them in the data structures of *WeightedInputSequences*, *Random InputSequences* and *Runtime Contexts*.  $N$  is the number of candidate runtime contexts. In our test, we devised 4 runtime contexts (including Non-background), so  $N = 4$ .  $p_{wgt}$  and  $P_{ctx}$  are the steering parameters for concretely

guiding the test.  $p_{wgt}$  is the probability of choosing a **weighted** input sequence for the upcoming test case. In practice, we initialize it as 50%, so the first test case has a chance of 50% to run with a weighted input sequence, and we will update and refine  $p_{wgt}$  dynamically during the entire testing. On the other hand, one element  $p_k$  in  $P_{ctx}$  represents the probability of choosing  $context_k$  in  $RuntimeContexts$ . We will also update  $P_{ctx}$  at runtime. The summation of elements in  $P_{ctx}$  is bound to 1.

$\Delta_{wgt}$  is the increment used to increase or decrease  $p_{wgt}$  to renew  $p_{wgt}$ .  $\Delta_{ctx}$  plays the same role for  $P_{ctx}$ . The larger  $\Delta_{wgt}$  and  $\Delta_{ctx}$  are, the more aggressively we tune the test direction.

**Details of the algorithm.** We first prepare data and initialize parameters ( $p_{wgt}$ ,  $P_{ctx}$ ,  $\Delta_{wgt}$  and  $\Delta_{ctx}$ ). We then start a test case, clear unrelated threads and data, and launch the app from root screen. Next, we decide the type of input sequence; weighted input sequences have a probability of  $p_{wgt}$  to be chosen. If the chosen type is “weighted”, we then select an unexplored sequence with the highest weight in  $WeightedInputSequences$ . Otherwise, we randomly select a sequence from  $RandomInputSequences$ . Likewise, for runtime context, we select one from  $RuntimeContexts$  with its corresponding probability.

We then feed the app with the selected input sequence and runtime context, and measure the device power. The power trace will be analysed to confirm whether an energy issue occurs. If there exists an energy issue, it implies that this type of input sequence and runtime context may be profitable for provoking more energy issues, our testing framework then steers lightly in this direction. Specifically, if it is triggered with a weighted input sequence, we increase  $p_{wgt}$  by  $\Delta_{wgt}$ . And after being increased,  $p_{wgt}$  should not exceed  $wgt\_up\_threshold$ . If it is a random input sequence, we decrease  $p_{wgt}$  by  $\Delta_{wgt}$ . Also, we keep  $p_{wgt} \geq wgt\_down\_threshold$ .

An analogous approach is applied to refining  $P_{ctx}$ . We check under which runtime context (e.g.  $context_k$ ) the issue occurs, then increase its testing probability (e.g.  $p_k$ ). However, the precondition is that there should be at least one element (except  $p_k$  itself) in  $P_{ctx}$  that are no less than  $ctx\_down\_threshold + \Delta_{ctx}$ , because on one hand, we intend to rebalance the probabilities, and on the other, we should let all contexts have at least a possibility of  $ctx\_down\_threshold$  to be tested.

### 5.3 Identify Energy Issues from Power Trace

We divide the power trace into five stages, i.e. PRE-OFF, IDLE, EXECUTION, BACKGROUND and SCREEN-OFF. This division can help us identify three types of energy issues: execution issues (including issues caused by unnecessary workload and excessively frequent operations), background issues (i.e. wasted background processing) and no-sleep issues. Figure 4 shows an illustration of power traces with these three types of energy issues.

PRE-OFF stage is the beginning stage where the device is powered but the screen is off. Then, the test case will be transferred to IDLE stage by turning on the screen. To enter EXECUTION stage, the subject application will be opened and run with a certain input sequence and runtime context, which are selected as shown in Section 5.2. After EXECUTION stage, the application will be fed with a press of *Home* button to enter BACKGROUND stage. The final stage is SCREEN-OFF stage, which begins when screen is supposed to be

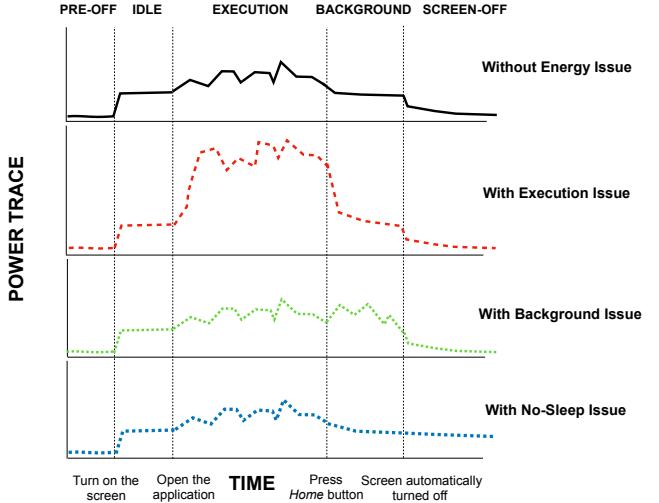


Figure 4: An illustration of power traces with energy issues.

turned off automatically, however, part of energy issues keep the screen on even at SCREEN-OFF stage, eating battery power badly.

**5.3.1 Identifying Execution Issues.** For execution issues, as we discussed in Section 1, evaluating the necessity of app workload is crucial for identifying them. Specifically, we employ the Dbscan clustering algorithm [8] (Density based spatial clustering of applications with noise) to fulfil this purpose. The objective of Dbscan is to classify multidimensional data points into three groups, namely, core points, border points and outlier points. After clustering, the data points should have the following properties: 1) For a core point, the number of its neighbours (the points within a range of  $\epsilon$  from it) is no less than a certain value,  $MinPts$ . Generally speaking, the core points are “quite close and gathered”. 2) For a border point, its neighbours are less than  $MinPts$ , but it is a neighbour of at least one core point or another border point. 3) For an outlier point, its neighbours are less than  $MinPts$ , and it does not have either a core or a border neighbour.

We treat each test case as a data point, and treat test cases in the same app category as a data set for clustering. The dimensions of each data point we employed for clustering are  $l_{chpp}$ ,  $n_{chpp}$ ,  $\mu_{chpp}$ ,  $\mu_{exe}$ , which are all extracted from power trace of EXECUTION stage of each test case.  $l_{chpp}$  is the total length of continuous-high-power periods. Continuous-high-power period is when power continuously exceeds a certain threshold longer than a certain length.  $n_{chpp}$  is number of these periods.  $\mu_{chpp}$  is average power of these periods.  $\mu_{exe}$  is average power of the entire EXECUTION stage. Dbscan then classifies the test cases into those three groups. We label test cases in core and border groups as “normal”, and the ones in outlier group as suspects for suffering from execution issues.

The motivation of this approach to probing for execution issues is this: usually “normal” energy use is sufficient to guarantee quality of user experience (QoE) of apps, outlier-level energy use is suspiciously problematic and unnecessary. And different app categories usually have distinct “normal” energy use (e.g. games vs. productivity apps). So we handle different app categories separately as shown above.

**Table 2: Technique package. The bold items are our originally-designed techniques, the italic are inspired by [2].**

Preparation before testing	Steer test direction
<ul style="list-style-type: none"> <li>• Candidate input sequences           <ul style="list-style-type: none"> <li>◦ Weighted input sequences               <ul style="list-style-type: none"> <li>- System APIs (I/O components)</li> <li>- Control-flow operations (CPU)</li> </ul> </li> <li>◦ Random input sequences</li> </ul> </li> <li>• Candidate runtime contexts           <ul style="list-style-type: none"> <li>◦ Normal ◦ Non-background</li> <li>◦ Flight Mode ◦ Network fail</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Balance weighted and random input sequences</li> <li>• Balance different runtime contexts</li> <li>• Improve efficacy of issue-detection</li> </ul>
<b>Identify energy issues from power trace</b>	
<ul style="list-style-type: none"> <li>• Identify execution issues           <ul style="list-style-type: none"> <li>◦ Identify background and no-sleep issues</li> <li>◦ Select dimensions</li> <li>◦ Cluster ◦ Label outliers</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>◦ Dissimilarity analysis</li> </ul>

Note that, the ultimate clustering model is available only when all test traces are collected. So at the beginning of testing, such a model is inaccessible since we have no power traces to build it. So, we need to decide, from which point during testing, we should start building a model using the collected data. To make our design reasonable, we did a pilot study by running 100 randomly-chosen test cases. We extracted the key features from the power traces and visualized the features. We found two “obvious” outliers using Dbscan. With further manual verification, both outliers were confirmed to be real issues.

As a density-based clustering algorithm, Dbscan may generate biased-models when outliers are of significant proportion in the dataset. In our problem domain, the number of outliers (abnormally high energy use) is naturally low, as shown in our pilot study and in our final evaluation (Section 6). Hence, in our experiment, we at first collected 50 power traces to bootstrap the model building process and then iteratively added new power traces to the dataset. As the dataset grows, the model becomes more accurate and powerful for identifying outliers. At last, we used the final model to recheck all power traces for issue detection.

Later, our evaluation on 89 apps (involving 35600 test cases) shows that only 1.1% test cases are outliers. Our framework detected 47 candidate execution issues from those 1.1% test cases. Only three (out of the 47) are false positives, indicating the high reliability of this approach. In contrast, current technology [2] detected 3 candidate execution issues from 30 apps, and still one of them is a false positive.

**5.3.2 Identifying Background and No-Sleep Issues.** If the app is free from background issues, the power trace in BACKGROUND stage is supposed to be similar to that in IDLE stage. We thus compute the dissimilarity value of the two traces. If the value is above a certain threshold (40% in our experiment), we label this test case as a candidate for a background issue.

We identify the no-sleep issues in the same way. We compare PRE-OFF with SCREEN-OFF. If the dissimilarity outstrips a certain threshold (50% in our experiment), we speculate this test case is suffering from a candidate no-sleep issue.

#### 5.4 Manual Verification

After the candidate issues are found, we manually verify whether they are actual energy issues. We adopt three criteria for distinguishing a real issue from a false positive. First, the energy cost

is not from the OS. Second, the energy cost is larger than normal cases by at least 10%. Third, user experience can be improved after removing the workloads. For the third criterion, which may be subjective, we clarify it using an example: an app is downloading large files for functional purposes, such as maps in games, which causes abnormally high energy cost. In this case, there is no easy solutions to save battery power since users may accept the expensive downloading process (removing it will affect app functionality). We then identify it as a false positive. However, for cases such as the issue in Leisure (as shown later in Table 3), where GIF animations are played when they are invisible, we identify it as a real issue since animations in such cases do not generate user-observable benefits. One can optimize the apps by removing such expensive computation. However, we agree that evaluating necessity of workload is an obvious challenge. There is no very effective and general solution yet.

#### 5.5 Comparison with the Most Relevant Work

Table 2 lists the differences between our technology and the most relevant work [2]: 1) For preparation before testing, since their work [2] only takes I/O components into account, their technology is impracticable for the important CPU-bound apps (e.g. games) and CPU-related energy issues. They also neglected the special runtime contexts which can trigger 25.0% of energy issues. 2) W.r.t steering test direction, their work does not have this feature because they only consider two dimensions of testing space, i.e. I/O-related input sequences and Normal context, whereas our framework additionally tests five more dimensions including CPU-related and random input sequences, and three more runtime contexts. So our searching space for energy issues is exponentially enlarged; we thus designed practical online steering strategy to balance different kinds of inputs and contexts, and improve the issue-detection efficacy. 3) For identifying energy issues from power trace, as we mentioned in Section 1, E/U theory [2] can hardly address execution issues (64.5% of all energy issues), by comparison, our framework employs advanced machine learning algorithm to analyse the energy use of apps and filter out these issues.

Benefiting from these targeting designs, our framework largely outperforms the state-of-art, as shown in Section 6.

### 6 EXPERIMENTAL EVALUATION

In this section, we first introduce our experimental setup. Then we evaluate our testing framework on various aspects, such as its efficacy in detecting energy issues, its comparison with the state-of-the-art, etc. The result shows that our testing framework largely outperforms current technology, showing the benefits both of our sound empirical study and our dynamic targeting techniques.

#### 6.1 Experimental Setup

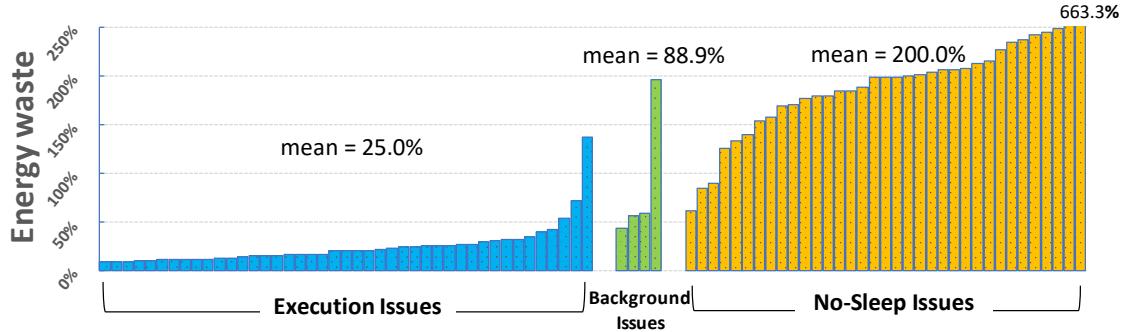
We employ the Odroid-XU4 development board<sup>11</sup>, whose processor has four big cores with a frequency of 2 GHz and four small cores with a frequency of 1.3 GHz. The board possesses a powerful 3D accelerator, Mali-T628 MP6 GPU. The high capacity of Odroid-XU4

<sup>11</sup><https://wiki.odroid.com/odroid-xu4/odroid-xu4>

**Table 3: Examples of detected energy issues.**

Application	Category	Issue Type	Activity or Symptom	Hit Rate <sup>1</sup>	Context or Non-background	Energy Waste	Reported before?
Leisure	News	Execution	3 GIFs playing on one page	1.0%	Normal	25.9%	No
GPS Status	Travel	Execution	Not obvious	47.0%	Normal	15.3%	No
BatteryDog	Tools	Execution	Editing lengthy text	1.0%	Normal	30.8%	No
Rocket Chat	Comm.	Execution	Connect to server	1.0%	Normal	12.8%	No
Chess Clock	Tools	Background	Device heated up	100.0%	WiFi Fail	59.2%	No
Vanilla	Multimedia	No Sleep	Enqueue many tracks	59.0%	WiFi Fail	242.8%	No
cgeo	Travel	No Sleep	App get stuck	2.0%	Flight Mode	179.4%	Yes
AntennaPod	Multimedia	No Sleep	Keep popping up messages	1.0%	Non-background	184.4%	Yes

1. Hit rate here is the percentage of test cases detected having the energy issue in that app.

**Figure 5: The energy waste of detected issues of different types**

board guarantees its performance for most applications on the market. It is also equipped with a power monitor, Smartpower2<sup>12</sup>, to measure the real-time power consumption. The sampling rate is 100 Hz. To assess its measurement variability, we randomly chose 20 test cases and ran each for 10 times. We thus collected 10 records of average power consumption for each test case. We employ coefficient of variation ( $C_v$ ) to indicate the variability. Specifically,  $C_v = \frac{\sigma}{\mu}$ , where  $\sigma$  and  $\mu$  are the standard deviation and mean of the 10 records for each test case. At last, the mean of the 20  $C_v$ s is about 0.8%, meaning a low measurement variability. Due to these rich and solid features, the Odroid board is widely employed in the field of energy optimization for mobile devices [41, 52, 57].

We use Android 4.4 KitKat as our target OS. Android is open-sourced and it captures around 74.13%<sup>13</sup> of the worldwide mobile OS market by Dec 2019. We evaluate our framework on the 89 app subjects. 27 of them have issue reports, 62 do no, as we introduced in Section 2. The **considerations** of employing these 89 apps for the evaluation are these: 1) Our framework is inspired by issue reports from the 27 apps, so it may be inapplicable to new apps, we thus employ the 62 subjects without issue reports as **new** apps to evaluate the generality of our framework. 2) Referring to the issue reports, we also can check whether our framework is capable of detecting unreported issues. 3) These apps are popular, well-maintained and of much higher quality than the apps adopted by previous research.

Our total testing time for the 89 app subjects is 2373.3 hours, i.e. 98.9 days, which were evenly spent on each app. (i.e. 1.11 days for

one app). Note that, the time of preparing weighted input sequences is also included, which takes 18.8% of the entire testing time.

## 6.2 The Efficacy of Our Testing Framework

The experimental result shows that our test detected 91 candidate energy issues, among which we manually confirmed **83 real energy issues**. 22.9% (19 out of 83) of these issues are from the 27 old apps, 77.1% (64 out of 83) are from the 62 new apps. After manual verification, we found **8 false positives**. Table 3 shows 8 examples of the detected energy issues. For instance, in Leisure, three animated GIFs are loaded and are played at the bottom of a certain page even though they are invisible to users most of the time. This execution issue wastes 25.9% energy use. It can be fixed by freezing the animation when the GIF pictures are not shown on the screen. For another example, when Chess Clock is not in use and backgrounded, the device will heat up from 41.2°C to 60.9°C due to the inefficient and long utilization of CPU. The average power of this issue is 59.2% higher than that of the IDLE stage.

Figure 5 shows the energy waste of detected energy issues. Energy waste is calculated using Equation (2).

$$w = (\frac{e_x}{e_n} - 1) \times 100\% \quad (2)$$

$w$  is the energy waste of the issue,  $e_x$  is average power of the corresponding stage in the test case with the corresponding issue (EXECUTION stage for execution issues, BACKGROUND stage for background issues, SCREEN-OFF stage for no-sleep issues).  $e_n$  is “normal” energy cost. We define “normal” energy cost individually for different issues. For background and no-sleep issues, we adopt average power at IDLE and PRE-OFF stage as normal cost, respectively. For execution issues, we use mean value of average

<sup>12</sup><https://www.odroid.co.uk/odroid-smart-power-2>

<sup>13</sup><http://gs.statcounter.com/os-market-share/mobile/worldwide>



**Figure 6: Comparison on issue-detection efficacy ( $r_s^d$ ) with the present technology.**

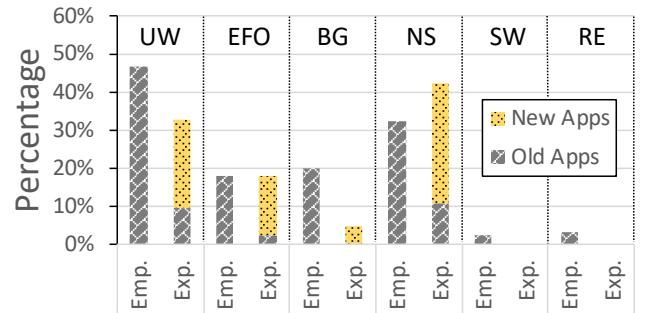
powers of EXECUTION stage in test cases in the same app category, as normal cost.

The experimental result shows that the energy waste of execution issues is 25.0% on average and up to 137.6%; the energy waste of background issues is 88.9% on average and at maximum 196.2%; the values for no-sleep issues are 200.0% and 663.3%, respectively. Overall, the average energy waste of all the issues is 101.7%.

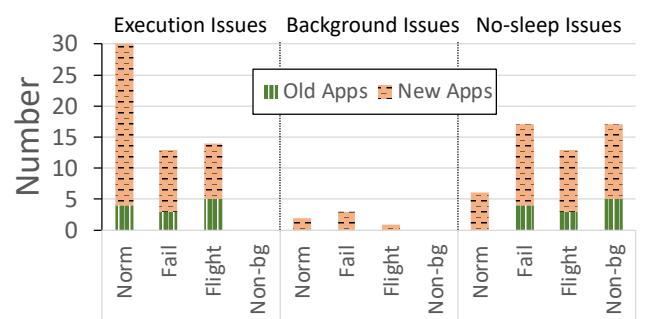
91.6% (76 out of 83) detected energy issues in our test are **newly-reported**, which on average double energy consumption of the apps. Without our tests, these serious energy issues might have never been detected even though they cause serious battery drain.

On the other hand, 95.9% (164 out of 171) energy issues in our empirical study are not listed in the issues detected by our test. The major reasons are as follows: Firstly, our standard of determining energy issues is much higher than that of developers. The issues detected in our test have outlier-level impacts on energy use; the energy wastage is usually above 10.0%. However, many issues detected by developers only cause small transient workloads, energy overuse can hardly reach 10.0%. For instance, Firefox issue 1057247 lets app re-fetch the failed favicon every 20 min, which is believed costly. So developers reduce the re-fetching frequency to conserve energy. For another example, VoiceAudioBookPlayer issue 299 makes the app scan material files everytime the user starts and leaves app. Developers then designed a smarter scanner to lessen the number of scans. However, these issues can not be noticed by our framework because they have very marginal influence on the metrics (i.e.  $l_{chpp}$ ,  $n_{chpp}$ ,  $\mu_{chpp}$  and  $\mu_{exe}$ ) we chose to identify execution issues. Secondly, a number (35.4%, 58 out of 164) of the issues are not reproducible, so our test cannot trigger them either. Thirdly, the variety of input sequences and runtime contexts in our test is not large enough to cover all of them due to the time limit.

The first and third reasons also shed light on **false negatives** of our test. The first reason implies that the issues that draw developers' attention but consume non-significant power (compared with our detected issues) can be missed by our test. It is because only outlier-level energy cost is deemed as an energy issue in our test. Loosening this strict criterion can help significantly reduce false negatives but may lead to false positives. The dilemma is caused by the difficulty of objectively and automatically judging the necessity of workloads. There is no general solution for this problem yet. Nevertheless, our framework can be flexibly configured to detect more energy issues if users are willing to tolerate some false positives. W.r.t. the third reason, since we were testing 89 apps, one



**Figure 7: Issue causes in empirical study and experiment.**



**Figure 8: Energy issues manifested under different contexts.**

day's test for one app results in three months' test for all apps. Developers can test their own app more comprehensively to realize larger coverage.

We now compare the efficacy of our testing framework with current technology [2]. We use the efficacy ratio ( $r_s^d$ ), i.e. the ratio of the number of detected issues to the number of subject apps, to indicate the efficacy of a framework. As shown in Figure 6, for execution issues, their  $r_s^d$  is 6.7% (2 issues out of 30 apps), ours is  $r_s^d$  is 49.4% (44 out of 89); for background (BG) and no-sleep (NS) issues together, their  $r_s^d$  is 33.3% (10 out of 30), while ours is 43.8% (39 out of 89). We combine background and no-sleep issues since their work did not distinguish them. The result confirms that our framework can detect a much larger number of more serious energy issues in higher-quality apps in comparison with the state-of-the-art. This is due to the fact that our work is based on the insightful empirical findings, rather than ungrounded assumptions.

### 6.3 Issue Cause and Manifestation

Figure 7 demonstrates breakdown of energy issues of different causes in empirical study and experiment. “Emp.” is experiment, “Exp.” is empirical study. Our experiment is conducted on both new and old apps, we thus plot them with different colours and patterns. “UW” is unnecessary workload, “EFO” is excessively frequent operations, “BG” is wasted background processing, “NS” is no-sleep, “SW” is spike workload, “RE” is runtime exception. As shown in Section 6.2, the issues detected in experiment are much more costly than those in empirical study. So this result indicates that, no matter for “big” or “small” issues, UW and EFO are always the very significant root causes, which justifies our Finding 1. And these issues are exactly the issues that current technology can hardly address.

Figure 8 demonstrates the number of energy issues triggered under different contexts in our experiment, which captures more detailed manifestation characteristics of energy issues. “Norm” is Normal, “Fail” is Network Fail, “Flight” is Flight Mode, “Non-bg” is Non-background. We can see that most execution issues are manifested in the Normal context because many scenarios where issues occur require normal network and GPS context. For example, as we discussed above, the issue in Leisure showed up only when those three GIF pictures were downloaded and showing on the page. We only have six ( $6/83 = 7.2\%$ ) background issues, which indicates that OS is competent in clearing potential bad influence of backgrounded apps at BACKGROUND stage. However, backgrounded apps may still suffer from no-sleep issues: Normal, Network Fail and Flight Mode provoke 6, 17, 13 no-sleep issues respectively. Furthermore, even though Non-background and Normal run in the same context, the former tends to provoke more no-sleep issues. In our test, it induces 17 issues. This result implies that special contexts (and Non-background) tend to incur anomalous behaviours of apps, such as bad use of wake-lock, and thus cause more no-sleep issues.

Figure 8 also shows that 37.3% (31 out of 83) energy issues can only be triggered under Network Fail and Flight Mode. This confirms Finding 2: special contexts, such as network fail, hide a significant number of serious energy issues.

## 7 RELATED WORK

Our work is related to multiple lines of research work. We will discuss three aspects: *understanding energy issues*; *detecting and diagnosing energy issues*; *fixing energy issues and optimizing software*.

**Understanding Energy Issues.** The style of empirical study that mines the data in repositories has been widely applied. For example, to characterize performance issues, a large body of research has been done for PC and server side software [17, 37, 55]. The first empirical study on characteristics of energy issues in the system of mobile device was done by Pathak et al. [39]. They mined over 39,000 posts from four online mobile user forums and mobile OS bug repositories, and studied the categorization and manifestation of energy issues. Xiao et al. [32] conducted a similar survey on three Android forums. The above studies involve issues in multiple layers across the system stack of mobile devices, from hardware, OS to applications. And the issue reports adopted in above studies were mostly proposed by end-users and random developers, who can hardly contribute very insightful understanding of the issues (e.g. root causes). The most related study is conducted by Liu et al. [30], investigating performance issues (including energy issues, as how they treated) reported in issue-tracking systems maintained by developers who developed the apps. However, the number of the studied energy issues is very small compared with our study. In summary, our study is targeted at app-level energy issues, and our employed issue reports are documented by professional developers of the corresponding high-quality projects. Our findings are more insightful and comprehensive.

**Detecting and Diagnosing Energy Issues.** Non-energy issues (e.g. security [43], compatibility [24] and performance [38] issues) can be detected plainly by analysing program artefacts. In contrast, to detect energy issues, researchers have to first learn the energy

characteristics of mobile devices and apps. Hence, researchers use OS, hardware and even battery features as predictors to infer energy information at device, component, virtual machine or application level [5, 18, 19, 21, 40, 44, 51, 54]. Shuai et al. [11] and Ding et al. [23] proposed approaches to obtaining energy information at source line level. The former requires the specific energy profile of the target systems. The latter utilizes advanced measurement techniques to obtain source line energy cost.

Two pieces of work [15, 16] from Jabbarvand et al. are close to our work. Our work differs from theirs in three main aspects. First, their works mainly address the challenges of issue manifestation (how to trigger the issues), while our work addresses the challenges of both issue manifestation and detection (how to identify the real existence of the triggered issues). Second, they assume that the over-use of certain APIs is the main source of energy issues, which is also assumed by [2]. However, APIs cannot cover all usage of CPU, which is a main energy consumer [10] on smartphones. In contrast, our work analyzes the CPU usage by tracing control-flow operations at code level. Third, their evaluations show the efficacy of their techniques for triggering previously-reported energy issues, but the efficacy for triggering previously-unknown issues is not justified due to the lack of issue-detection mechanisms.

The work of Banerjee et al. [2] is the most relevant to ours. Two key differences prevent it from uncovering most serious energy issues detected in our test. The first is that we have deeper understanding of energy issues, which helped us address the execution issues and special contexts that they can not handle. The second is technical difference, as we elaborated in Section 5.5: we took CPU overuse into account when pre-designing input sequences, we developed online test-steering strategy to explore the vast searching space, we employed advanced machine learning algorithm to identify execution issues from power trace, etc. In summary, owing to our careful empirical study and well-designed testing technology, our framework surpasses the state-of-the-art in detecting all kinds of energy issues.

**Fixing Energy Issues and Optimizing Software.** A large amount of research effort on energy-saving for mobile devices has been focused on the main hardware components, such as the CPU, display and network interface. The CPU-related techniques involve dynamic voltage and frequency scaling [7], heterogeneous architecture [9, 25, 27] and computation offloading [20, 48]. Techniques targeting the display include dynamic frame rate tuning [13], dynamic resolution tuning [12] and tone-mapping based back-light scaling [1, 14]. Network-related techniques try to exploit idle and deep sleep opportunities [28, 47], shape the traffic patterns [6, 42], trade-off energy against other design-criteria [4, 45, 53], etc. Such work attempts to reduce energy dissipation by optimizing the hardware usage; there are also several pieces of work aiming at designing new hardware and devices [49, 50, 56]. Besides, another line of research work is dedicated to solving background and no-sleep issues [5, 29, 35].

Two pieces of work [3, 34] provide systematic approaches to optimizing software source code for energy-efficiency. In the former, Boddy et al. attempted to decrease the energy consumption of software by handling code as if it were genetic material so as to evolve to be more energy-efficient. In the latter, Irene et al. proposed

a framework to optimize Java applications by iteratively searching for more energy-saving implementations in the design space.

## 8 CONCLUSION

In this paper, we conducted an empirical study on software energy issues in 27 well-maintained open-source mobile apps. Our study revealed root causes and manifestation of energy issues. Inspired by this study, we fully implemented a novel testing framework for detecting energy issues. It first statically analyses the source code of app subjects and then extracts the candidate input-sequences with large probability of causing energy issues. We also devised several artificial runtime contexts that can expose deeply-hidden energy issues. Our framework effectively examines apps with the inputs and contexts under a systematic scheme, and then automatically identifies energy issues from power traces. A large-scale experimental evaluation showed that our framework is capable of detecting a large number of energy issues, most of which existing techniques cannot handle. These issues on average double the energy cost of the apps. Finally, we showed how developers can utilize our test reports to fix the issues.

## ACKNOWLEDGEMENT

This research was supported in part by the China NSFC Grant (61902249, 61802164, 61872248 and 61702343), Guangdong NSF 2017A030312008, Shenzhen Science and Technology Foundation (No. ZDSYS20190902092853047), Guangdong Science and Technology Foundation (2019B111103001, 2019B020209001), GDUPS (2015). Kaishun Wu is the corresponding author.

## REFERENCES

- [1] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.
- [2] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 588–598, New York, NY, USA, 2014. ACM.
- [3] B. R. Bruce, J. Petke, and M. Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.
- [4] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking energy-performance trade-off in mobile web page loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 14–26, New York, NY, USA, 2015. ACM.
- [5] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 40–52, New York, NY, USA, 2015. ACM.
- [6] C. Chiasserini and R. Rao. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on*, 19(7):1385–1394, Jul 2001.
- [7] V. Devadas and H. Aydin. On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers*, 61(1):31–44, Jan 2012.
- [8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.
- [9] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.
- [10] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76, 2016.
- [11] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.
- [12] S. He, Y. Liu, and H. Zhou. Optimizing smartphone power consumption through dynamic resolution scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, pages 27–39, New York, NY, USA, 2015. ACM.
- [13] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song. Raven: Perception-aware optimization of power consumption for mobile games. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 422–434, New York, NY, USA, 2017. ACM.
- [14] A. Iranli and M. Pedram. DTM: Dynamic tone mapping for backlight scaling. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 612–617, New York, NY, USA, 2005. ACM.
- [15] R. Jabbarvand, J.-W. Lin, and S. Malek. Search-based energy testing of android. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1119–1130. IEEE Press, 2019.
- [16] R. Jabbarvand and S. Malek. Mdroid: An energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 208–219, New York, NY, USA, 2017. Association for Computing Machinery.
- [17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.
- [18] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.
- [19] J. Koo, K. Lee, W. Lee, Y. Park, and S. Choi. Batttracker: Enabling energy awareness for smartphone using li-ion battery characteristics. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.
- [20] K. Kumar and Y. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.
- [21] S. Lee, C. Yoon, and H. Cha. User interaction-based profiling system for android application tuning. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 289–299, New York, NY, USA, 2014. ACM.
- [22] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM.
- [23] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.
- [24] L. Li, T. F. Bissyande, H. Wang, and J. Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 153–163, New York, NY, USA, 2018. ACM.
- [25] X. Li, G. Chen, and W. Wen. Energy-efficient execution for repetitive app usages on big.little architectures. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 44:1–44:6, New York, NY, USA, 2017. ACM.
- [26] X. Li and J. P. Gallagher. A source-level energy optimization framework for mobile applications. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 31–40, Oct 2016.
- [27] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *SIGPLAN Not.*, 47(4):13–24, Mar. 2012.
- [28] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 146–159, New York, NY, USA, 2008. ACM.
- [29] Y. Liu, C. Xu, S. Cheung, and V. Terragni. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE 2016, 2016.
- [30] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.
- [31] Y. Liu, C. Xu, S. C. Cheung, and J. Laijij. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sep. 2014.
- [32] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Presented as part of the 10th USENIX Symposium on Networked*

- Systems Design and Implementation (NSDI 13)*, pages 57–70, Lombard, IL, 2013. USENIX.
- [33] A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda. Picsel: Measuring user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 70–79, New York, NY, USA, 2008. ACM.
- [34] I. Manotas, L. Pollock, and J. Clause. Seeds: A software engineer’s energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.
- [35] M. Martins, J. Cappos, and R. Fonseca. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 563–575, Santa Clara, CA, 2015. USENIX Association.
- [36] A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*, pages 260–269, Nov 2003.
- [37] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR ’13, pages 237–246, Piscataway, NJ, USA, 2013. IEEE Press.
- [38] A. Nistor and L. Ravindranath. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 282–292, New York, NY, USA, 2014. ACM.
- [39] A. Pathak, Y. C. Hu, and M. Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 5:1–5:6, New York, NY, USA, 2011. ACM.
- [40] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys ’12, pages 29–42, New York, NY, USA, 2012. ACM.
- [41] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference on*, pages 1–6, 2014.
- [42] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 48–55, May 2004.
- [43] L. Qiu, Y. Wang, and J. Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 176–186, New York, NY, USA, 2018. ACM.
- [44] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, pages 107–120, Berkeley, CA, USA, 2012. USENIX Association.
- [45] A. Sehati and M. Ghaderi. Energy-delay tradeoff for request bundling on smartphones. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.
- [46] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.
- [47] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys ’05, pages 261–274, New York, NY, USA, 2005. ACM.
- [48] K. Sucipto, D. Chatzopoulos, S. KostaÅs, and P. Hui. Keep your nice friends close, but your rich friends closer â€¢ computation offloading using nfc. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.
- [49] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger. A 90nm low-power FPGA for battery-powered applications. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA ’06, pages 3–11, New York, NY, USA, 2006. ACM.
- [50] L. Wang, M. French, A. Davoodi, and D. Agarwal. FPGA dynamic power minimization through placement and routing constraints. *EURASIP J. Embedded Syst.*, 2006(1):7–7, Jan. 2006.
- [51] F. Xu, Y. Liu, Q. Li, and Y. Zhang. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 43–55, Lombard, IL, 2013. USENIX.
- [52] H. Yamamoto, T. Hirano, K. Muto, H. Mikami, T. Goto, D. Hillenbrand, M. Takamura, K. Kimura, and H. Kasahara. Oscar compiler controlled multicore power reduction on android platform. In *Languages and Compilers for Parallel Computing*, pages 155–168, Cham, 2014. Springer International Publishing.
- [53] Z. Yan and C. W. Chen. Rnb: Rate and brightness adaptation for rate-distortion-energy tradeoff in http adaptive streaming over mobile devices. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom ’16, pages 308–319, New York, NY, USA, 2016. ACM.
- [54] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC’12, pages 36–36, Berkeley, CA, USA, 2012. USENIX Association.
- [55] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR ’12, pages 199–208, Piscataway, NJ, USA, 2012. IEEE Press.
- [56] L. Zhong and N. K. Jha. Energy efficiency of handheld computer interfaces: Limits, characterization and practice. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys ’05, pages 247–260, New York, NY, USA, 2005. ACM.
- [57] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, Feb 2015.

# Automated Classification of Actions in Bug Reports of Mobile Apps

Hui Liu\*

School of Computer Science and Technology, Beijing  
Institute of Technology  
Beijing, China  
liuhui08@bit.edu.cn

Jiahao Jin

School of Computer Science and Technology, Beijing  
Institute of Technology  
Beijing, China  
3220180713@bit.edu.cn

## ABSTRACT

When users encounter problems with mobile apps, they may commit such problems to developers as bug reports. To facilitate the processing of bug reports, researchers proposed approaches to validate the reported issues automatically according to the *steps to reproduce* specified in bug reports. Although such approaches have achieved high success rate in reproducing the reported issues, they often rely on a predefined vocabulary to identify and classify actions in bug reports. However, such manually constructed vocabulary and classification have significant limitations. It is challenging for the vocabulary to cover all potential action words because users may describe the same action with different words. Besides that, classification of actions solely based on the action words could be inaccurate because the same action word, appearing in different contexts, may have different meaning and thus belongs to different action categories. To this end, in this paper we propose an automated approach, called *MaCa*, to identify and classify action words in Mobile apps' bug reports. For a given bug report, it first identifies action words based on natural language processing. For each of the resulting action words, *MaCa* extracts its contexts, i.e., its enclosing segment, the associated UI target, and the type of its target element by both natural language processing and static analysis of the associated app. The action word and its contexts are then fed into a machine learning based classifier that predicts the category of the given action word in the given context. To train the classifier, we manually labelled 1,202 actions words from 525 bug reports that are associated with 207 apps. Our evaluation results on manually labelled data suggested that *MaCa* was accurate with high accuracy varying from 95% to 96.7%. We also investigated to what extent

\*corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397355>

Mingzhu Shen

School of Computer Science and Technology, Beijing  
Institute of Technology  
Beijing, China  
3120181025@bit.edu.cn

Yanjie Jiang

School of Computer Science and Technology, Beijing  
Institute of Technology  
Beijing, China  
jiangyanjie@bit.edu.cn

*MaCa* could further improve existing approaches (i.e., Yakusu and ReCDroid) in reproducing bug reports. Our evaluation results suggested that integrating *MaCa* into existing approaches significantly improved the success rates of ReCDroid and Yakusu by 22.7% = (69.2%-56.4%)/56.4% and 22.9% = (62.7%-51%)/51%, respectively.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging;
- Computing methodologies → Machine learning algorithms.

## KEYWORDS

Mobile Testing, Classification, Test Case Generation, Bug report

### ACM Reference Format:

Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated Classification of Actions in Bug Reports of Mobile Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397355>

## 1 INTRODUCTION

Mobile applications are highly popular [31]. Apple store has 2.2 millions of mobile apps for download whereas Google Play has 2.8 millions. Such mobile apps, however, contain numerous defects that frustrate users [46]. When users encounter problems with such apps, they may commit such problems to developers as bug reports [9, 15] via bug tracking systems, like Bugzilla [1], Google Code Issue Tracker [4], and Github Issue Tracker [3]. To facilitate the resolution of the reported issues, such issue tracking systems often request and help users to specify detailed steps for reproducing the reported issues [20]. As a result, as suggested by existing empirical studies, such bug reports often contain necessary information in natural language for reproduction of the reported issues [15, 45].

Manual validation of the reported issues is often time consuming, and thus automated validation is highly desirable [15, 46]. Fazzini et al. [15] proposed an automated approach, called Yakusu, to translate bug reports into test cases for mobile apps. To build the mapping from textual bug reports to UI actions, Yakusu conducts lexicon normalization and object standardization on the textual bug

reports. Lexicon normalization is to replace non-standard words with their canonical form to simplify parsing and understanding of actions. The key of lexicon normalization is to replace action words with one of the standard action words (vocabulary) that are selected manually by analyzing a corpus of bug reports. For example, Yakusu replaces the action word "*Tap*" in "*Tap on the Publish button*" with "*Click*". Object standardization is to replace phrases that referring to target UI element with the ID of the corresponding UI element in app (called ontology in the paper). After the lexicon normalization and object standardization, Yakusu explores each clause in the bug report. If the root of a clause is a predefined action word, Yakusu extracts an abstract action for the clause based on the dependency tree of the clause. Otherwise, Yakusu computes the lexical similarity between the whole clause and properties of UI elements, and retrieves the most similar UI element. Finally, Yakusu generates test cases by mapping the resulting abstract actions to UI actions that are discovered by exploring mobile apps. More recently, Zhao et al. [46] proposed another automated approach, call ReCDroid, to automatically reproduce Android application crashes from bug reports. Similar to Yakusu, ReCDroid also employs a predefined (manually built) vocabulary of action words. ReCDroid classifies such action words into three categories, i.e., CLICK, EDIT, and GESTURE. ReCDroid identifies and classifies events/actions in bug reports by matching the reports against the predefined vocabulary. For different types of events, ReCDroid employs different grammar patterns to match the sentence, and creates abstract actions based on the matching patterns. Finally, ReCDroid maps the resulting abstract actions to UI actions by dynamic exploration of the associated app with dynamic ordered event tree.

Both Yakusu and ReCDroid are accurate, achieving high success rate in reproducing the reported issues. However, both Yakusu and ReCDroid rely on a predefined vocabulary to identify and classify actions in bug reports. Depending on manually constructed vocabulary (and classification) results in significant limitations. First, it is challenging for the vocabulary to cover all potential action words. Users may describe the same action with different words, i.e., new words out of the predefined vocabulary could be employed by users to describe existing actions. For example, "*Touch the Menu Button*", "*Press the Menu Button*", and "*Click the Menu Button*" employed different action words to represent the same action. Second, the classification of actions solely based on the action words could be inaccurate. The same action word, appearing in different contexts, may have different meaning and thus belongs to different action categories. For example, the same action word '*change*' has different meaning and belongs to different action categories in the following sentences: "*change orientation ...*" (belonging to category ROTATE), "*Change the dollar amount ...*" (belonging to category TYPE), and "*Change the language ...*" (belonging to category CLICK). Inaccurate identification and classification of action words could result in failure of the automated validation approaches like Yakusu and ReCDroid. For example, ReCDroid will apply improper patterns to match clauses if the action words are recognized or classified incorrectly, which in turn results in failure of action discovery.

To this end, in this paper we propose a Machine learning based automatic approach (called *MaCa*) to identifying and Classifying actions in Mobile apps' bug reports. For a given bug report, *MaCa* identifies action words based on natural language processing and

dependency trees. For each of the resulting action words, *MaCa* extracts its contexts, i.e., its enclosing segment, the associated UI target, and the type of its target element. The extraction is conducted based on both natural language processing and static analysis of the associated apps. The action word and its contexts are then fed into a machine learning based classifier that predicts the category of the given action word in the give contexts. To train the classifier, we manually labelled 1,202 actions words from 525 bug reports that are associated with 207 apps. The manual labelling classifies the actions words into the following five categories: *CLICK*, *TYPE*, *ROTATE*, *SWIPE*, and *SCROLL*. Our evaluation results on manually labelled data suggested that *MaCa* was accurate with high accuracy varying from 95% to 96.7%. We also investigated to what extent the proposed approach can further improve existing approaches (i.e., Yakusu and ReCDroid) in reproducing bug reports by integrating *MaCa* into existing approaches. Evaluation results suggested that integrating *MaCa* into existing approaches significantly improved the success rates of ReCDroid and Yakusu by 22.7% = (69.2%-56.4%)/56.4% and 22.9% = (62.7%-51%)/51%, respectively. Evaluation results also suggested that the proposed approach was accurate in identifying and classifying such action words (called *unseen words*) that did not appear in the training data.

In summary, our paper makes the following contributions:

- An automated technique for identifying and classifying actions in Mobile apps' bug reports written in natural languages.
- A publicly available implementation of the technique, and a manually labelled dataset for training of the approach [39].
- An empirical study providing initial evidence of the effectiveness of our approach and its potential in improving existing approaches in automated validation of bug reports. The replication package is available on Github [39].

## 2 MOTIVATING EXAMPLES

One reason for machine learning based automated classification of action words is that users may leverage different words to represent essentially the same actions. The following bug report (issue #2523 of app *MvvmCross* [7]) is a typical example:

1. **Select** the second tab on the Bottom Navigation View;
2. **Click** the button with the text 'Click Me';
3. **Press** the hardware back button

We notice that the reporter leverages three different words, i.e., "*select*", "*click*", and "*press*", to describe the same action (clicking UI elements). Such diversified description makes it challenging to collect and classify all potential action words manually.

Another reason for machine learning based classification of action words is that the same word may have different meaning in different contexts. For example, the word "*change*" in the following bug reports represents different actions:

- (1) Issue #783 of app *Lockwise*[6]: "... 4. **Change** the language to an unsupported one ...";
- (2) Issue #22 of app *LibreNew*[5]: "... **Change** server address to an invalid one, e.g., xxxyyzz ...";
- (3) Issue #34330 of app *Flutter*[2]: "... 3. **Change** orientation a few times ...".

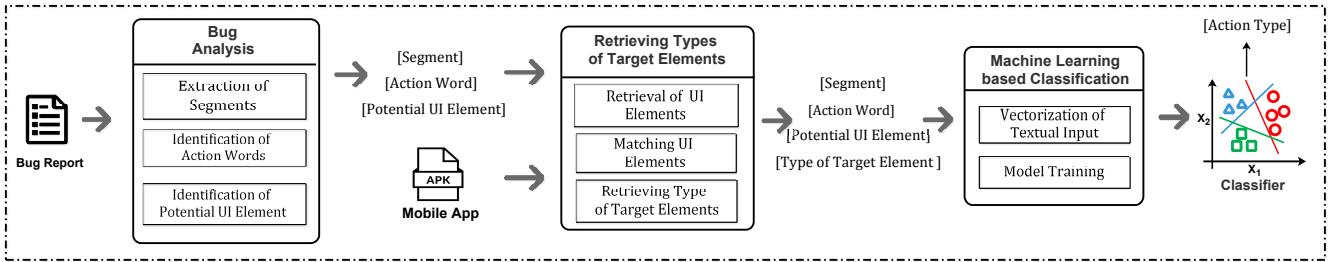


Figure 1: Overview

The first "change" [6] represents a click action to change/select a specific language from a given list. The second "change" [5] represents a typing action, e.g., to type in server address "xxxyyzz". The third "change" [2] represents rotation of the mobile device. Such polysemous words make it challenging to classify action words solely based on the words themselves. Additional information, like the contexts of the words, should be considered as well.

We conclude based on the preceding examples that it may result in significant limitations if we manually construct vocabularies of actions words and classify them based action words only. To this end, the proposed approach employs machine learning techniques to identify and classify actions, and thus has the potential to handle unknown/new action words because machine learning techniques have the potential to handle new items that are not contained in training data [18, 33]. Besides that, it leverages the contexts of action words for classification, and thus has the potential to classify polysemous words correctly.

### 3 APPROACH

#### 3.1 Overview

An overview of the proposed approach (MaCa) is presented in Fig. 1. The approach takes bug reports as input and finally generates a list of actions as well as their action types. Three main steps of the approach are bug analysis, retrieving types of target elements, and machine learning based classification.

*Bug analysis* decomposes steps to reproduce (notated as s2r) into segments that represent individual actions in reproduction of the reported issues. For example, s2r "Press a custom server, and input an invalid server address" should be decomposed into two segments "Press a custom server" and "input an invalid server address". For each of the resulting segments, the proposed approach leverages natural language processing techniques to identify action word, and potential UI elements associated with the action. For example, from segment "Press a custom server", we should identify action word "Press" and its associated potential UI element "a custom server".

*Retrieving types of target elements* takes input of an action word, its enclosing segment, its associated potential UI elements, and the associated mobile app. From the associated mobile app, the proposed approach leverages static analysis to identify all UI elements that may be created statically or dynamically by the app [42]. The proposed approach leverages word2vec [27] to encode properties of the resulting UI elements as well as the textual description (in bug report) of potential UI elements. From the resulting UI elements, the proposed approach leverages lexical similarity to identify most

likely target UI element for the given action, and returns the type of the target element. Taking the action word "Press" in segment "Press a custom server" for example, the proposed approach matches potential UI element ("a custom server") against all UI elements in app *NewsBlur*, which results in a matching UI element whose ID is *login\_custom\_server*. Consequently, the proposed approach returns the type of this element, i.e., *TextView*, as the type of the target element for action word "Press".

*Machine learning based classification* takes input of an action word, its enclosing segment, its associated potential UI elements, and the type of its target element. It leverages machine learning based classification techniques to predict the type of the action. Considering the significant imbalance of data, the proposed approach computes weights for each of the categories (labels), and optimizes the training process with the resulting weights. The proposed approach employs powerful BERT [14] to turn the textual input into a single numeric vector. Finally, it leverages a logistic regression model [10] to predict the action type of the resulting numeric vector. Notably, the training of the employed classifier requires labelled training data, and thus we manually created a training dataset based on reports from Github.

#### 3.2 Analyzing Bug Reports

We analyze bug reports to extract action words and their contexts from steps to reproduce (s2r). As the first step on the processing of s2r, we replace all hyperlinks in bug reports with a special symbol "*URL*" because the hyperlinks are often lengthy and detailed addresses (contents of the links) are often irrelevant to parsing of natural languages or the classification of actions. For example "Go on <https://download.lineageos.org/>" would be revised as "Go on *URL*".

After the preprocessing, we partition s2r into segments by *spaCy* [8], a widely used natural language processing toolkit. Basically, *spaCy* decomposes sentences according to punctuation marks, and the decomposition is often accurate. However, for clauses containing conjunctions, e.g., "and", the decomposition could be incomplete. For example, *spaCy* takes "Alternatively press the mail and from contextual menu click on delete" as a single segment although it should be partitioned into two segments: "Alternatively press the mail" and "from contextual menu click on delete". To this end, we further decompose the segments returned by *spaCy* according to a list of conjunctions, i.e., "and", "or", "after", and "before".

For each of the resulting segments, we parse it into a dependency tree by *spaCy* [8]. Two sample dependency trees are presented in Fig. 2 where the associated segments are "click your library" and

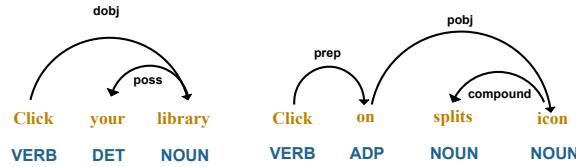


Figure 2: Dependency Trees of Segments

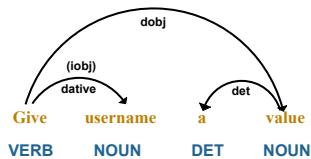


Figure 3: Dependency Tree and Indirect Object

"click on splits icon", respectively. From the dependency trees, we locate the verbs (the roots of the trees), e.g., "click" for the sample trees in Fig. 2. This verb is then taken as an action word. To locate the target UI elements associated with the discovered action word, we select the nouns/noun phrases with the following special syntactic dependency with the action word: direct object (dobj), preposition object (pobj), and indirect objects (iobj). Direct object (dobj), as shown on the left part of Fig. 2) is the noun phrase which is the accusative object of the verb. The object of a preposition (pobj, as shown on the right part of Fig. 2) is the noun or pronoun governed by a preposition. Pobj is usually the noun or pronoun to the right of the preposition. The indirect object (iobj) of a verb, as shown in Fig. 3, is any nominal phrase that is a core argument of the verb but is not its subject or direct object.

We collect all such nouns and noun phrases as the potential UI elements for the action. For the example on the left part of Fig. 2, the action word is "click" and its associated potential UI element is {"you library"}. For the example on the right part of Fig. 2, the action word "click" is associated with potential UI element {"splits icon"}.

### 3.3 Retrieving Types of Target Elements

In the preceding section, we extract action words as well as their associated potential UI elements. In this section, we search for the most likely UI element (notated as *actualElement*) in the associated app that the give action word *aw* may act on. Detailed algorithm is presented in Algorithm 1. Input of the algorithm includes the given action word *aw* and the associated mobile app (notated as *app*). Output of the algorithm is the element type of the target of the action word.

To retrieve the most likely UI element in the associated app that the action targets, we collect all UI elements that are created by the mobile app (Line 3). We analyze the resource configuration files (XML documents) and collect a set of UI elements that are created at design time [36]. Besides that, we also analyze the source code of the app to retrieve a set of UI elements that could be created by the app at runtime (with the help of Gator [42]). All such UI elements are returned and assigned to collection *UES*.

---

**Algorithm 1:** : Retrieving Type of Action Target

---

```

Input: aw; // action word
        app // the associated app
Output: ElementType // Type of the target UI element
1 maxSimilarity = 0 ;
2 actualElement = null ;
3 UES = RetrieveUIElements(app) ;
4 for each pElement in aw.PotentialUIelements do
5   for each element in UES do
6     similarity = ComputeSim(element, pElement+aw) ;
7     if similarity > maxSimilarity then
8       maxSimilarity = similarity ;
9       actualElement = element ;
10      end
11    end
12    appSim = Sim(app.getName(), pElement+aw);
13    if appSim > maxSimilarity then
14      maxSimilarity = appSim;
15      actualElement = app;
16    end
17  end
18  if actualElement ≠ null AND maxSimilarity > β then
19    | ElementType = actualElement.type;
20  else
21    | ElementType = null ;
22  end
23 return ElementType

```

---

The algorithm then goes into the main iteration (Lines 4-17). On each iteration (Lines 5-16), the algorithm selects the most likely UI element in the associate app for a given potential UI element (notated as *pElement*) associated with the action word *aw*. The selection is based on the lexical similarity between *pElement* and UI elements within the app. The ID and display text (notated as *dText*) of UI element (notated as *element*) are leveraged for the computation of lexical similarity:

$$\text{ComputeSim}(\text{element}, \text{pElement} + \text{aw}) = \max(\text{Sim}(\text{element.ID}, \text{pElement} + \text{aw}), \text{Sim}(\text{element.dText}, \text{pElement} + \text{aw})) \quad (1)$$

The action word *aw* (a verb) is appended to *pElement* for similarity computation because developers may name UI elements with verbs on some special cases. Method *Sim* computes the lexical similarity between two texts by representing them as digital vectors (also known as embedding) by well-known word2vec [27], and computing the cosine similarity of the two vectors. Notably, word2vec converts a single word into a vector, and thus we present the whole text as the average of the vectors for words within the text. On Line 12, the algorithm also computes the similarity between the potential UI element and the name of the associated app. It is likely that some users may describe the launch of the app as the first step to reproduce, and in this case the target element is the app itself.

The two iterations select the most likely UI element as the target of the action word, noted as *actualElement*. If its lexical similarity with one of the potential UI element descriptions is higher than  $\beta$ ,

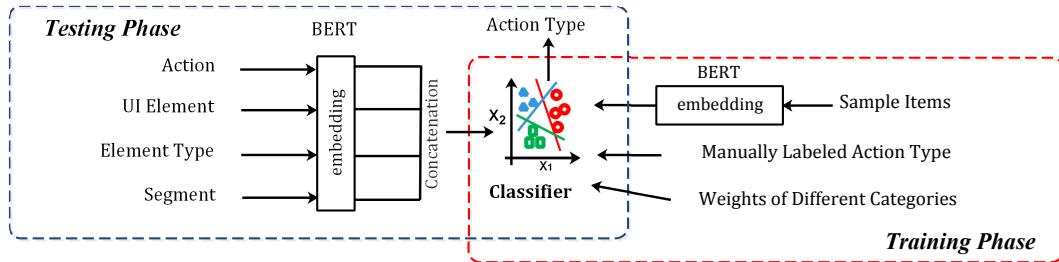


Figure 4: Machine Learning Based Classification

the selection succeeds and the type of *actualElement* is returned (Lines 18–19). Otherwise, the selection fails and the algorithm returns a special symbol *null*. Notably, we empirically set the minimal similarity  $\beta$  to 0.5 that is frequently employed as default values in machine learning based classifiers.

### 3.4 Machine Learning Based Classification

The last part of the approach is a machine learning based classifier that classifies an action word *aw* into one of the categories  $CaS = \{c_1, c_2 \dots c_n\}$  where  $n$  is the total number of categories. The structure of the classifier and the training/testing process are presented in Fig. 4.

Although we have tried to retrieve the types of target elements (as specified in Section 3.3), a machine learning based classifier is still needed because of the following reasons. First, the retrieved type of the target element alone could be insufficient to determine the action type. For example, on “*TextView*” users may perform CLICK or SCROLL (even more action types on the latest iPhone). Consequently, we cannot determine the action type solely based on the type of associated target element. Second, there is no guarantee that the static analysis can successfully retrieve the type of the target element. The second step of the approach (“retrieving types of target elements”) returns NULL when it fails to retrieve the target element based on the textual description of UI elements in S2R. It is challenging to match UI elements according to the textual description of UI elements. This is also the major reason for existing approaches (like Yakusu and ReCDroid) to recognize/classify action words first and then match associated UI elements according to the identified action types (not the other way around).

It is likely that the data (both training data and testing data) are not balanced, i.e., some categories are significantly larger than others. However, unbalanced data will cause machine learning models to focus on major categories (with the larger numbers of items) and to undervalue other minor categories. Machine learning based classifiers often leverage the weights (distribution) of different categories to cope the imbalance [38, 41]. To this end, we compute the weights of each category as follows:

$$wt_i = \frac{1}{\log(ItemCount(c_i))} \quad (2)$$

where  $wt_i$  is the weight for category  $c_i$ , and  $ItemCount(c_i)$  is the number of times belonging to category  $c_i$ . With the resulting weights  $wt$ , we compute the loss of the classifier as the weighted average of the loss for individual items during the training phase.

According to the definition of the weights, the smaller a category is, the larger its weight would be. Consequently, the weighted loss enlarges the penalty for the misclassification on items belonging to small categories.

Input of the classifier is a tuple:

$$\begin{aligned} item = & < aw, EnclosingSegment(aw), UIelements(aw), \\ & ElementType(aw) > \end{aligned} \quad (3)$$

*aw* is an action word extracted in Section 3.2, *EnclosingSegment(aw)* is the segment where *aw* is extracted, *UIelements(aw)* is a list of potential UI elements associated with *aw* (as extracted in Section 3.2), and *ElementType(aw)* is the type of the most likely target of *aw*.

The input *item* is essentially a sequence of texts, and such texts should be vectorized before they could be fed into classifiers. We convert each part of the *item* into a vector by BERT (Bidirectional Encoder Representations from Transformers) [14] (as shown in the embedding layer of Fig. 4). BERT was proposed recently by researchers from Google, and has been proved effective in vectorizing texts [17, 21, 35]. After converting each of the texts into a numeric vector, we concatenate the resulting vectors as a longer one that is finally fed into the machine learning based classifier:

$$\begin{aligned} numericInput = & [BERT(aw), BERT(EnclosingSegment(aw)), \\ & BERT(UIelements(aw)), BERT(ElementType(aw))] \end{aligned} \quad (4)$$

For the machine learning based classifier, we empirically employ a logistic regression model [10]. As revealed in Section 4, logistic regression results in higher accuracy of the classification than other common machine learning techniques, including Naive Bayesian [23], Random Forest [24], Support Vector Machine [40], AdaBoost [34], Multi-Layer Perceptron [30], Convolutional Neural Network [22], Long Short-Term Memory [19], and Bi-directional Long Short-Term Memory [37]. Notably, the proposed approach is not confined to logistic regression, and it can work with any classification techniques that label numeric vectors with one of the predefined labels. The selected classifier employs a *softmax* function that could be formalized as follows:

$$P(x; \theta) = \frac{1}{\sum_{i=1}^n e^{w_i \cdot x + b_i}} \begin{bmatrix} e^{w_1 \cdot x + b_1} \\ e^{w_2 \cdot x + b_2} \\ \vdots \\ e^{w_n \cdot x + b_n} \end{bmatrix} \quad (5)$$

A training process should optimize the parameters  $w = < w_1, w_2 \dots w_n >$  and  $b = < b_1, b_2 \dots b_n >$  on training data. After training, the model would compute the probability that item  $x = numericInput$

belongs to  $j$ th category as follows:

$$P(y = j|x) = \frac{e^{w_j \cdot x + b_j}}{\sum_{i=1}^n e^{w_i \cdot x + b_i}} \quad (6)$$

## 4 EVALUATION

In this section, we evaluate the proposed approach by investigating the following research questions:

- **RQ1:** Is *MaCa* accurate in classifying actions in Mobile apps' bug reports?
- **RQ2:** Is *MaCa* accurate in identifying and classifying action words that are unseen in training data?
- **RQ3:** To what extent do different classification techniques affect the performance of *MaCa*?
- **RQ4:** To what extent do the contexts of action words affect the performance of *MaCa*?
- **RQ5:** To what extent can *MaCa* improve the performance of automated bug report validation by integrating it into existing approaches (i.e., ReCDroid and Yakusu)?
- **RQ6:** Does *MaCa* outperform keyword match method based on pre-defined vocabulary?

### 4.1 Dataset

The dataset is composed of three parts. The first part of the dataset is the 51 bug reports created by Zhao et al. [46] for the evaluation of ReCDroid, notated as *icseData*. They randomly crawled 300 bug reports containing keywords "crash" and "exception" from Github, included 15 bug reports from FUSION [28], and 25 bug reports from the 62 reproducible bug reports collected by Fazzini et al. [15]. They applied manual filtration on such bug reports to exclude non-reproducible bug reports as well as those that did not result in actual app crashes. Finally, 51 bug reports were selected for their evaluation.

The second part of the dataset is the 39 bug reports collected by Fazzini et al. [15], notated as *isstaData*. Fazzini et al. [15] retrieved issues from Github using keywords "android", "crash", "reproduce", and "version". From the resulting issues, they randomly selected 100 for manual analysis, and finally picked up 62 reproducible bug reports for the evaluation of Yakusu. However, only 39 of them were still available and reproducible on the day we accessed it, and thus *isstaData* includes 39 bug reports only.

The third part, notated as *newData*, contains 525 bug reports of mobile apps. We collected such bug reports from Github that contain "Android (or iOS)" and "reproduce". We manually checked the resulting top 2,000 bug reports, and manually excluded those that did not contain steps to reproduce and those that have been included by either *icseData* or *isstaData*. Finally, 525 bug reports containing useful steps to reproduce were kept for the evaluation. For each of the bug reports, the first two authors manually identified steps to reproduce, and manually classified the action words in such steps. We also manually labelled the UI types for those whose associated Apps are not available. The participants achieved high agreement on the classification, resulting in a high Cohen's Kappa coefficient [13] of 0.7866 that suggests excellent inter-rater agreement.

## 4.2 Experiment Design

**4.2.1 RQ1: Accuracy of MaCa:** To measure the accuracy of *MaCa*, we conducted ten-fold cross validation on *newData* that we created manually. We partitioned the 525 bug reports in *newData* into ten groups with approximately equal size (either 52 or 53). On each fold, we selected one group as the testing data whereas the others were taken as training data. Notably, each group was taken as testing data for once. We trained *MaCa* on the selected training data, applied the resulting classifier to the testing data, and computed the accuracy of the classification on the testing data:

$$\text{accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of items in testing data}} \quad (7)$$

where a prediction was correct only if the predicted action type was exactly the same as manually labelled. The denominator of the equation was the total number of items instead of the number of predictions. Consequently, action words that *MaCa* failed to identify (and thus fails to classify) was counted in by the denominator.

Besides accuracy, we also computed some common performance metrics for multiple-classification, i.e., precision and recall for each category [32], and *macro F1* [25] over the whole dataset. Notably, for our task (where each of the items would be assigned a single label), *micro F1* equals accuracy, and thus we do not present *micro F1* in the rest of the paper.

To further evaluate its performance, we also conducted cross-dataset validation: we trained *MaCa* with the *newData* and evaluated the resulting *MaCa* on *icseData* and *isstaData*, respectively.

**4.2.2 RQ2: Unseen Words:** One of the weakness of predefined vocabulary and classification is that they cannot identify or classify new action words outside the vocabulary (we call them unseen words). To investigate how well *MaCa* can handle such unseen words, we identified the rarest actions words (notated as *Raws*) in *NewData*, removed bug reports containing such words (resulting in a new data *NewData'*), trained *MaCa* with *NewData'* from scratch, and applied the resulting *MaCa* to *Raws*.

**4.2.3 RQ3: Effect of Classification Techniques:** As specified in Section 3.4, the proposed approach depends on classification techniques to predict action types. Although the approach selects logistic regression as the default classification technique, the approach itself is open to any of the classification techniques that can label a numeric vector with one of the predefined labels. Consequently, we investigated how different classification techniques may affect the performance of the proposed approach, and whether the default setting, i.e., logistic regression, results in the best performance. To this end, we replaced the logistic regression-based classifier in *MaCa* with different classifiers, and repeated the evaluation as introduced in Section 4.2.1. More specifically, we have tried the following classification techniques: Random Forest [24], Support Vector Machine (SVM) [40], Long Short-Term Memory (LSTM) [19], Bi-directional Long Short-Term Memory (Bi-LSTM) [37], Convolutional Neural Network (CNN) [22], Multi-Layer Perceptron (MLP) [30], Naive Bayesian [23], and AdaBoost [34]). Such techniques were selected because they are accurate and have been widely used.

**4.2.4 RQ4: Effect of Contexts:** As specified in Section 3.4, *MaCa* leverages various contexts (i.e., its target UI element, type of its

**Table 1: Performance of MaCa (Ten-Fold Cross-Validation)**

Metrics	1#	2#	3#	4#	5#	6#	7#	8#	9#	10#	Avg
Accuracy	97.5%	96.7%	96.7%	94.2%	96.7%	95.8%	99.2%	97.5%	95.0%	96.7%	96.6%
Macro F1	92.7%	94.8%	95.2%	92.1%	86.7%	91.5%	92.0%	89.6%	86.2%	93.1%	91.4%

target UI element, and its enclosing segment) of the action word to be classified. To investigate to what extent such contexts may affect the performance of the classification, we employed different contexts of the words and repeated the ten-fold evaluation on *newData* (as specified in Section 4.2.1). Notably, as introduced in Section 3.4, the contexts of an action word include its enclosing segment, its potential UI elements, and the type of the target. Consequently, we tried all of the possible combination of such contexts to evaluate the effect of different parts of the contexts.

**4.2.5 RQ5: Effect on Existing Approaches:** To investigate to what extent *MaCa* can improve the performance of ReCDroid and Yakusu, we integrated *MaCa* into ReCDroid and Yakusu, respectively. The evaluation on Yakusu was conducted as follows:

- First, we applied Yakusu to dataset *icseData* that Zhao et al. [46] manually built to evaluate ReCDroid, and computed its performance (success rate).
- Second, we applied *MaCa* to dataset *icseData*. *MaCa* classified the action words in *icseData*. Based on the classification, we replaced the out-of-vocabulary action words in *icseData* with their equivalent words (i.e., words belonging to the same category) from the vocabulary of Yakusu. The updated dataset was denoted as *icseData'*.
- Third, we applied Yakusu to *icseData'* and computed its performance (success rate).

Notably, to evaluate the effect of *MaCa* on Yakusu, we leveraged dataset *icseData* instead of *newData* or *isstaData* because of the following reasons. First, *newData* may contain reports that do not result in crashes. As a result, we could not apply Yakusu to *newData* because Yakusu requires bug reports that are associated with crash issues. Second, we did not leverage *isstaData* because this dataset had been specially created for the evaluation of Yakusu, and all of the action words within this dataset had been manually added to the predefined vocabulary of Yakusu. The evaluation on ReCDroid was done similarly on dataset *isstaData* that Fazzini et al. [15] manually created to evaluate Yakusu.

**4.2.6 RQ6: Compared against Static Vocabulary based Approach:** An intuitive alternative to the proposed approach is to match (and classify) action words according to a pre-defined vocabulary (we call it *vocabulary-based approach*). To compare *MaCa* against vocabulary-based approach, we conducted the following evaluation:

- We partitioned the 525 bug reports in *newData* into ten groups with approximately equal size (either 52 or 53), and conducted ten-fold cross-validation on the resulting ten groups.
- On each fold, we selected one group as the testing data whereas the others were taken as training data. We built a static vocabulary of action words according to the action words (and their labels) in the training data. If a single word

**Table 2: Results of Cross-Dataset Evaluation**

Evaluation Dataset	newData (Ten-Fold)	isstaData	icseData
Metrics			
Accuracy	96.6%	96.7%	95.0%
Macro F1	91.4%	91.1%	89.8%

(appearing in different reports) belonged to different categories, we classified it into the category it most frequently belonged to. With the resulting vocabulary, we classified action words in the testing data and compared the classification against manual labeling. We also trained *MaCa* with the training data and tested it with the testing data.

### 4.3 Results and Analysis

**4.3.1 RQ1: MaCa Is Accurate:** Results of the ten-fold cross validation are presented in Table 1. From this table, we make the following observations:

- First, *MaCa* was highly accurate. The accuracy varied from 94.2% to 99.2%, with an average of 96.6%. The high accuracy suggested that *MaCa* succeeded frequently in predicting action types. We also noticed that *MaCa* achieved a high Macro F1 score as well: the average *macro F1* was up to 91.4%.
- Second, *MaCa* was stable. In different folds, the evaluation data were partitioned (into training and testing datasets) in different ways. However, the accuracy varied slightly from 94.2% to 99.2%, with a small standard deviation ( $\sigma$ ) of 0.014. The same was true for *macro F1* whose standard deviation was 0.031.

The results of the cross-dataset evaluation also suggested that *MaCa* was accurate. The results are presented in Table 2. From this table, we observe that training *MaCa* with *newData* and testing the resulting *MaCa* on *isstaData* resulted in high accuracy of 96.7% and high F1 of 91.1%. Such performance was comparable with the within-dataset ten-fold cross-validation (i.e., the performance presented in the second column of Table 2). Evaluation results on *icseData* (the last column of Table 2) also confirmed the conclusion that *MaCa* was accurate.

We also analyzed the performance of *MaCa* on different categories. On the evaluation data (notated as *newData*), there were five different categories/labels: *CLICK*, *TYPE*, *ROTATE*, *SWIPE*, and *SCROLL*. The performance on different categories was presented in Table 3. From this table, we make the following observations:

- First, *MaCa* was accurate for all of the categories. The accuracy varied from 92.7% (*TYPE*) to 99.4% (*SWIPE*). The minimal precision, recall, and F1 score were 87.9% (*ROTATE*), 81.9% (*SCROLL*), and 87.1% (*SCROLL*), respectively.

**Table 3: Performance on Different Categories**

Metrics	CLICK	TYPE	ROTATE	SWIPE	SCROLL
Accuracy	94.3%	92.7%	98.2%	99.4%	98.5%
Precision	96.1%	90.5%	87.9%	95.2%	93.1%
Recall	95.9%	93.9%	89.9%	90.9%	81.9%
F1	96.0%	92.2%	88.9%	93.0%	87.1%

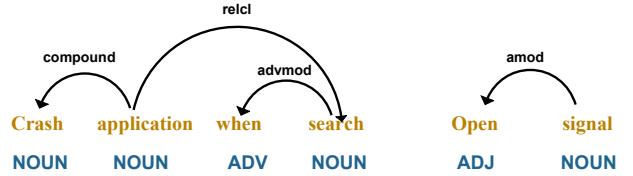
- Second, MaCa achieved the best performance on *CLICK* and the lowest performance on *SCROLL*. One possible reason for the relatively low performance (especially recall) on *SCROLL* is the small size of this category. Only 32 out of the 1,202 actions belong to this category, accounting for 2.7% of the actions in the dataset. It is challenging for the employed classifier to learn effective features of the category from such a small number of training items. In contrast, *CLICK* accounts for 70.4% of the actions in the dataset, which resulted in much bigger training data (thus better performance).

We noticed that the evaluation data were not evenly distributed over different categories. 70.4% of the items in the dataset belong to *CLICK*, 21% belong to *TYPE*, 3.1% belong to *ROTATE*, 2.8% belong to *SWIPE*, and 2.7% belong to *SCROLL*. The first two categories, i.e., *CLICK* and *TYPE*, are much more popular than others. To this end, the proposed approach assigns different weights to different categories while computing loss of the model in training phase (as introduced in Section 3.4). To investigate the effect of such weights, we removed the weights and repeated the evaluation. Evaluation results suggested that the average accuracy reduced significantly from 96.6% to 86.1%, and the *macro F1* also reduced significantly from 91.4% to 76.4%. Table 4 presents its performance on different categories, where cells in gray represent increased performance (compared to Table 3) and others represent decreased performance. By comparing this table (where weights are disabled) against Table 3 (where weights are provided), we make the following observations:

- Disabling the weights resulted in significant reduction in accuracy on all of the categories. The reduction varied from 5.6% (*CLICK*) to 20% (*ROTATE*).
- Disabling the weights also resulted in similar reduction in precision and F1 scores.
- Concerning the recall of MaCa, disabling the weights had significantly different effect on different categories. On one side, it increased recall on the largest category (*CLICK*) slightly from 95.9% to 97.1%. On the other side, it significantly reduced recall one *ROTATE* (from 89.9% to 33.3%), *SWIPE* (from 90.9% to 81.8%), and *SCROLL* (from 81.9% to 66.7%). For the second biggest category (*TYPE*) that accounted for 21% of the whole data, disabling weights did not result in significant change in recall (93.9% vs. 92.8%). One possible reason for the difference is the significant difference in their size: *CLICK* accounts for 70.4% of the whole data whereas *ROTATE*, *SWIPE* and *SCROLL* all together account for less than ten percentage of the whole data. Without weights, the classifier would be significantly bias for the dominating category (e.g., *CLICK*), i.e., more likely to predict items as *CLICK*.

**Table 4: Performance of MaCa (Without Weights)**

Metrics	CLICK	TYPE	ROTATE	SWIPE	SCROLL
Accuracy	88.7%	86.3%	78.2%	84.9%	92.3%
Precision	89.6%	90.5%	64.2%	89.9%	68.7%
Recall	97.1%	92.8%	33.3%	81.8%	66.7%
F1	93.2%	91.6%	43.9%	85.7%	67.7%

**Figure 5: Missed Action Words**

Based on the analysis, we conclude that providing the weights for different categories leads to significant increase of performance on smaller categories whereas the performance on the largest categories could be slightly reduced.

To reveal the reasons for failed cases, we manually analyzed the actions that MaCa failed to classify correctly. Analysis results suggest that one of the major reasons for the failure is that the employed NLP toolkit failed to identify action words. Two typical examples are presented in Fig. 5. For the example on the left, action word "search" was recognized as a noun. For the example on the right, the action word "open" was recognized as an adjective. In total, such incorrect identification of action words led to 9 out of the 40 failed cases.

Based on the preceding analysis, we conclude that the proposed approach is accurate. With improvement in natural language processing, the proposed approach could be further improved.

**4.3.2 RQ2: Succeed on Unseen Words:** From *newData*, we identified 20 rarest action words, removed bug records containing any of these words from *newData*, trained MaCa from scratch with the resulting shrunken dataset *newData'*, and applied MaCa to bug reports containing the 20 rarest words. The rarest words had been removed from *newData'*, and thus they were unseen words for MaCa trained on *newData'*. Such unseen words belonged to different categories: 11 *CLICK*, 8 *TYPE*, and 1 *SWIPE*. Eighteen of them appeared once whereas the other two appeared twice in *newData*.

Evaluation results suggested that MaCa was accurate in identifying and classifying unseen action words. On 22 steps to reproduce (s2r) containing unseen tokens, MaCa accurately identified and classified 20 action words, resulting in a high accuracy of 90.9% = 20/22.

We conclude based on the preceding analysis that MaCa has the potential to identify and classify unseen action words, which is a significant advantage over predefined static vocabulary.

**4.3.3 RQ3: Influence of Classification Techniques:** To investigate to what extent the employed classification techniques can affect the performance of the proposed approach, we replaced the logistic regression based classifier with other common classification techniques and repeated the evaluation. Evaluation results are presented in Table 5. The first column presents different classification

**Table 5: Influence of Classification Techniques**

Techniques	Dataset	newData		isstaData		icseData	
		Accuracy	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1
Logistic Regression ( <i>Default</i> )		96.6%	91.4%	96.7%	91.1%	95.0%	89.8%
LSTM		89.8%	81.7%	85.9%	79.4%	88.3%	81.7%
Bi-LSTM		89.1%	81.3%	89.4%	83.7%	87.4%	81.9%
MLP		86.0%	43.5%	84.1%	54.6%	82.9%	50.2%
SVM		85.0%	56.6%	82.6%	57.8%	85.6%	61.4%
Random Forest		83.4%	80.1%	83.5%	79.9%	81.7%	78.6%
Naive Bayes		80.8%	72.3%	78.8%	68.3%	79.9%	72.3%
AdaBoost		78.4%	20.6%	80.2%	33.4%	75.2%	30.3%
CNN		45.6%	37.2%	48.4%	39.7%	52.3%	44.1%

**Table 6: Effect of Contexts**

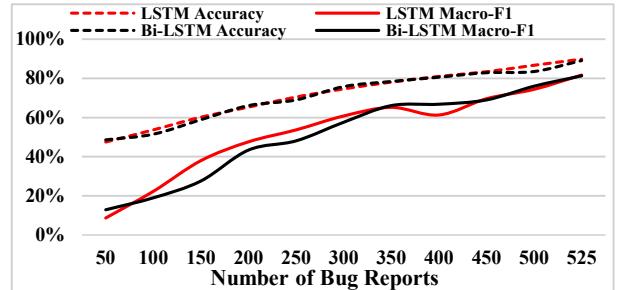
	Potential UI Elements	Type of Target Element	Enclosing Segment	newData		isstaData		icseData	
				Accuracy	Macro F1	Accuracy	Macro F1	Accuracy	Macro F1
1	✓	✓	✓	96.6%	91.4%	96.7%	91.1%	95.0%	89.8%
2	✓	✓		90.6%	49.3%	91.0%	60.4%	92.7%	57.7%
3	✓		✓	82.4%	29.7%	82.3%	28.5%	80.1%	29.6%
4		✓	✓	90.2%	41.3%	88.5%	59.3%	86.4%	54.8%
5	✓			77.9%	17.9%	73.4%	24.9%	75.3%	20.4%
6		✓		84.5%	36.6%	84.7%	39.4%	83.6%	38.3%
7			✓	80.8%	24.9%	80.3%	22.6%	77.6%	20.5%
8				71.4%	13.8%	71.2%	17.6%	69.3%	19.7%

techniques where the first one is logistic regression (default setting of MaCa). The other columns present performance of the proposed approach when different classification techniques were employed to replace the default setting. Notably, we evaluated the influence of classification techniques in different datasets, i.e., *newData* (ten-fold cross-validation, on the second and third columns), *isstaData* (the fourth and fifth columns), and *icseData* (the last two columns).

From Table 5, we make the following observations:

- First, the default setting (i.e., logistic regression) resulted in the best performance. On *newData*, it led to the highest accuracy (96.6%) as well as the highest *Macro F1* (91.4%). On *isstaData* and *icseData*, logistic regression also resulted in the highest performance.
- Second, although LSTM and Bi-LSTM resulted in significantly lower performance compared to logistic regression, they significantly outperformed other classification techniques, i.e., Random Forest, SVM, CNN, MLP, Naive Bayesian, and AdaBoost. On one side, LSTM and its variant Bi-LSTM are good at handling texts whereas the input of the classifiers was essential a sequence of texts. Consequently, both LSTM and Bi-LSTM led to good performance. On the other side, LSTM and Bi-LSTM often require a large number of training data that were not available for our task. As a result, they failed to outperform logistic regression that had weaker requirements on the size of training data. In future, however, if we can expand the training dataset manually or automatically, LSTM and Bi-LSTM have the potential to outperform logistic regression.

To validate the potential of LSTM and Bi-LSTM on our task, in Fig. 6 we depict the quantitative relation between size of labelled

**Figure 6: Potential of LSTM-based Classifiers**

data and the performance of the proposed approach (on *newData*). From this figure, we observe that the performance increases stably with the increased data size.

We conclude based on the preceding analysis that the default setting (logistic regression) outperformed alternative classification techniques for our task. However, with the increase of training data, other techniques, e.g., LSTM and Bi-LSTM, have the potential to further improve the performance.

**4.3.4 RQ4: Contexts Are Critical:** To reveal the effect of different contexts of the action words, we employed different combination of the contexts, i.e., the enclosing segment, the potential UI elements, and the type of target element. Results of the evaluation are presented in Table 6. Columns 2-4 presents the absence or presence of different contexts. The other columns present the performance of the proposed approach with the selected contexts on given testing data (i.e., *newData*, *isstaData*, and *icseData*, respectively). Each row

**Table 7: MaCa Enhanced Existing Approaches**

Approaches	#Bug Reports where the approach succeeded	Success Rate
Yakusu	26	50.98%
Yakusu+MaCa	32	62.7%
ReCDroid	22	56.41%
ReCDroid+MaCa	27	69.2%

represents a unique combination of the contexts. From the table, we make the following observations:

- Disabling any of the contexts resulted in significant reduction in performance. Compared to the default setting where all of the contexts were presented, other combinations resulted in significant reduction in both accuracy and *macro F1*.
- Type of target element was critical for the performance of the proposed approach. For example, on *newData*, disabling it decreased accuracy significantly from 96.6% to 82.4% and *macro F1* from 91.4% to 29.7%. Disabling other contexts, i.e., the enclosing segment and the potential UI elements, resulted in smaller, but still significant, reduction in performance.
- Effect on *macro F1* was more significant than that on accuracy (that was equal to *micro F1*). The reason is that the data were imbalanced, and thus significant changes in performance on small categories would not significantly change overall performance (e.g., accuracy). However, such significant changes in performance on small categories had obvious effect on *macro F1* that was the arithmetic mean of *F1* scores over different categories.

We conclude based on the preceding analysis that all of the selected contexts, i.e., the enclosing segment, the potential UI elements, and the type of target element, are useful for the classification. Disabling any of them would result in significant reduction in performance.

**4.3.5 RQ5: MaCa Enhanced Existing Approaches:** As an initial application of the proposed approach, we applied it to the automated validation of bug reports by integrating it into existing bug report validation approaches, i.e., Yakusu and ReCDroid. Evaluation results are presented in Table 7. The first column of the table presents the evaluated approaches. *Yakusu* and *ReCDroid* represent the original approaches proposed by Fazzini et al. [15] and Zhao et al. [46], respectively. *Yakusu + MaCa* and *ReCDroid + MaCa* represent approaches enhanced by MaCa. Notably, Yakusu and ReCDroid are evaluated on different datasets, and thus their denominators (i.e., the total numbers of bug reports) in calculating percentages (success rates) are different. As a result, the higher absolute number of Yakusu does not necessarily lead to a higher percentage (success rate) because Yakusu is applied to the larger dataset. As specified in Section 4.2, Yakusu and Yakusu+MaCa are evaluated on *icseData* that were created by Zhao et al [46] to evaluate ReCDroid. In contrast, ReCDroid and ReCDroid+MaCa are evaluated on *isstaData* that were created by Fazzini et al. [15] to evaluate Yakusu.

From Table 7, we observe that *MaCa* significantly improved the performance of both Yakusu and ReCDroid. It improved the

**Table 8: Bug Reports Where MaCa Rescued Yakusus**

	App	Issue No.	Out-of-vocabulary action words	Failure
1	Olam	1	tipying	No matching
2	LibreNews	22	Change	No matching
3	LibreNews	27	Change	Incorrect matching
4	ventriloид	1	left	Incorrect matching
5	Obd-Reader	22	start	No matching
6	Markor	194	–	action word "Insert" is misclassified

success rate of Yakusu from 50.9% to 62.7%, with a relative increase of  $22.9\% = (62.7\% - 50.9\%) / 50.9\%$ . The same was true for ReCDroid whose success rate was improved significantly by MaCa from 56.41% to 69.2%, resulting in a relative increase of  $22.7\% = (69.2\% - 56.41\%) / 56.41\%$ . The evaluation results provide initial evidence of the usefulness of the proposed approach.

To further reveal the reasons for the success of MaCa, we manually analyzed the bug reports where the original approaches (i.e., Yakusu or ReCDroid) failed but the enhanced approaches (i.e., *Yakusu + MaCa* or *ReCDroid + MaCa*) succeeded. Table 8 presents the details of the six bug reports where MaCa rescued Yakusu. From this table, we observe that such bug reports come from five different apps. Five out of the six bug reports contain action words that are out of the predefined vocabulary of Yakusu (called out-of-vocabulary action words). If a s2r does not contain predefined action words (defined by the vocabulary), Yakusu leverages semantic comparison between s2r and the elements in the ontology of the relevant app to match UI elements. However, compared to the action word based heuristics, such semantic based matching is much more risky. As shown in Table 8, the semantic based matching for the six bug reports either failed to match any UI elements or resulted in incorrect matching (i.e., wrong UI elements). MaCa successfully identified such action words, classified them correctly, and replaced them with equivalent words from the given vocabulary. As a result, MaCa helped Yakusu generate test cases successfully for such bug reports. We take the third bug report of Table 8 (issue #27 of app *LibreNews*) as an example to explain how MaCa helped. The failed step was "*Change the server*." Because the action word "*Change*" was out of vocabulary, Yakusu compared the whole segment (i.e., "*Change the server*") against properties of ontology of the app *LibreNews*. The semantic similarity between the segment and the correct target element (notated as E2 whose ID is "*server\_url*") was 0.6187. In contrast, its similarity with another element (notated as E2 whose ID was "*server\_status*") was 0.6347. Consequently, Yakusu mapped the segment to wrong UI element (whose ID was "*server\_status*"), which resulted in failure on this bug report. MaCa classified the action as *typing*, and thus Yakusu searched for editable UI elements only as the potential target of the action. As a result, element E2 (a *TextView* component) was excluded because it was not editable. In contrast, the right one, E1 (an *EditText* component) was editable and thus it was finally retrieved (correctly) as the target of the action.

Table 9 presents the details of the five bug reports where MaCa rescued ReCDroid. From the table, we also observe that four out of the five involved bug reports contain out-of-vocabulary action

**Table 9: Bug Reports Where MaCa Rescued ReCDroid**

	App	Issue no.	Out-of-Vocabulary Action Words	Failure
1	K-9 mail	1910	expand	timeout
2	ODK Collect	360	specify	timeout
3	Kandroid	2	makes	timeout
4	K-9 mail	2540	configuring	ignored
5	BeHe ExploreR	35	–	action word "go" is misclassified

words. As a result, ReCDroid failed to reproduce the reported crashes. We take the fourth bug report (issue #2540 of app *K-9 mail*) in Table 9 to illustrate how MaCa helped. ReCDroid failed to reproduce the following step "*Choose the inbox after configuring an account*". ReCDroid took it as a single step because it contained a single action word "*choose*" (that came from the predefined vocabulary), and thus another action "*configuring an account*" was missed. MaCa identified these two action words successfully because it decomposed the sentence into two segments according to conjunction "*after*" and identified "*configuring*" as a typing action. As a result, it helped ReCDroid reproduce both of the steps.

The last bug report on Table 9 (issue #35 of app *BeHe ExploreR*) does not contain out-of-vocabulary action words. The bug report says "*Go on https://download.lineageos.org/. Try to download a rom. The app will crash*". The action word "*go*" was classified as CLICK in the vocabulary. However, in this case, the word represented a typing action instead of a click action: Users should type in the URL ended with an *Enter* key. As a result, ReCDroid failed to reproduce this step because of the misclassification. MaCa accurately classified the action word based on its contexts, replaced it with "*type*", and thus helped ReCDroid reproduce this step. The same is true for the last bug report on Table 8 where "*Insert*" in s2r "*Insert image*" was misclassified by the vocabulary as TYPE whereas MaCa classified it correctly as CLICK.

Notably, one possible reason for the effect of MaCa on Yakusu and ReCDroid is that the training data of MaCa might contain the out-of-vocabulary action words listed in Table 8 and Table 9. For example, the vocabulary of ReCDroid did not contain keyword "*expand*" (and thus it failed on the first report of Table 9) whereas this word appeared in the training data for MaCa. To investigate this issue, from the training data, we removed all of the items containing any of the out-of-vocabulary action words on Table 9. After that, we re-trained MaCa from scratch with the clean training data, and applied the resulting MaCa to the first four bug reports on Table 9. Evaluation results suggested that MaCa successfully identified and classified all of the unseen/out-of-vocabulary action words in the bug reports. For bug reports in Table 8, we conducted similar evaluation, and results also suggested that MaCa successfully identified and classified all of the out-of-vocabulary action words on Table 8 even if such words had been removed from the training data. This is one of the strengths of machine learning based approaches that they may handle action words that have never been manually labelled in the training data.

**4.3.6 RQ6: Static Vocabulary based Approach:** As specified in Section 4.3.6, we compared MaCa against the alternative approach (called *vocabulary-based approach*) on *newData*. Our evaluation results suggest that *vocabulary-based approach* was significantly less accurate than *MaCa*. Its accuracy was 72.2%, significantly lower than that (96.6%) of *MaCa*. Its F1 (53.6%) was also significantly lower than that (91.4%) of *MaCa*.

One reason for the low accuracy of *vocabulary-based approach* is that new actions words are emerging, and thus pre-defined vocabularies may miss some new words. Another reason for the low accuracy is that the same action word may belong to different categories, depending on its contexts.

#### 4.4 Threats To Validity

The primary threat to external validity is the representativeness of the involved apps and bug reports. To reduce the threat, we crawled bug reports from Github, and reused bug reports collected by other researchers. The number (615) of involved bug reports is limited because of the complexity involved in manual analysis and labelling of the bug reports. Another reason for the limited number of bug reports is that only a small number of the publicly available bug reports explicitly specify steps to reproduce the reported issues. For example, among the 2,000 bug reports we manually analyzed, only 525 of them explicitly specify the reproduction steps.

The second threat to external validity is that only two existing approaches (i.e., Yakusu and ReCDroid) were involved to evaluate the effect of the proposed approach on automated validation of bug reports. The results could not be generalized to other approaches in automated validation of bug reports. To reduce the threat, we selected Yakusu and ReCDroid that represent the state of the art in this field. Another reason for the selection is that these two approaches came from different research groups.

A threat to constructive validity is that the manual labelling of the dataset could be inaccurate. The authors, instead of the developers who handled the reported issues, labelled the dataset because we failed to request the original developers to label the data. Without in-depth knowledge of the involved apps, however, the authors may misclassify some of the actions. To reduce the threat, the manual labelling was conducted by two authors with rich experience in app development and bug issue handling. A high Cohen's Kappa coefficient [13] of 0.7866 suggests excellent inter-rater agreement.

## 5 RELATED WORK

Analysis on bug reports for various tasks has been a hot research topic in the field of software engineering [15, 29, 46]. For space limitation, we only discuss such work that are most closely related to our technique.

The most closely related work is *Yakusu* [15] and *ReCDroid* [46]. Fazzini et al. [15] proposed an automated approach, called *Yakusu*, to translate Android bug reports into executable test cases. *Yakusu* parses textual segments into dependency trees and identifies abstract actions from the resulting trees according to a predefined vocabulary of UI actions. Finally, *Yakusu* generates executable test cases by mapping the abstract actions into a sequence of UI actions that are available for the given mobile app. Zhao et al. [46] proposed

another automated approach, called ReCDroid, to reproduce Android application crash from bug reports that are specified in natural language. Notably, our static analysis to retrieve types of target elements is inspired by the static analysis in ReCDroid [46]. ReCDroid differs from *Yakusu* in that ReCDroid extracts input values from bug reports for editable events whereas *Yakusu* generates random values for such events. Besides that, ReCDroid reproduces crashes directly whereas *Yakusu* generates test cases that could be executed (and modified if needed) to reproduce reported crashes. Both *Yakusu* and ReCDroid employ predefined (manually constructed) vocabularies of action words. As introduced in Section 1, such manually constructed vocabularies result in significant limitations in action recognition and classification. Our approach that could recognize and classify actions automatically helps such approaches to avoid predefined vocabularies. Evaluation results in Section 4 validated the effectiveness of our approach in enhancing *Yakusu* and ReCDroid.

Motwani and Brun [29] proposed an automatic approach, called Swami, to generate precise oracles and executable test cases from API specifications (i.e., ECMA-262 JavaScript specification). Different from the proposed approach, Swami takes the assumption that the textual input is structured natural language specifications. Based on this assumption, Swami identifies parts of the specification relevant to the implementation to be tested with a regular expression. The regular expression concerns the section number and the method name in the structured document. After that, Swami employs a series of regular expression-based rules to extract information of the syntax for the methods to be tested. Besides bug reports and API specification, various resources have been leveraged to generate test cases, e.g., police reports on self-driving cars [16], automatically generated error messages [44], app states [26], and log messages collected from the field [43].

Oscar and Carlos [11] proposed an approach, called Euler, to assess the quality of the steps to reproduce in bug reports. In case of ambiguous steps, steps described with unexpected vocabulary, and steps missing in the report, Euler provides actionable feedback to reporters who may improve the quality of bug reports according to the suggestions. As a preprocessing step of quality assessment, Euler identifies and classifies action words. Similar to *Yakusu* and ReCDroid, Euler employs a predefined vocabulary and determines the category of a given action word by comparing its lemma against those in the vocabulary. Our approach may further improve Euler in the same as it helps *Yakusu* and ReCDroid.

Oscar et al. [12] proposed an approach, called DEMIBuD, to identify missing information in bug descriptions. Among different implementations of DEMIBuD, DEMIBuD-ML is based on machine learning techniques. DEMIBuD-ML is essentially a linear Support Vector Machines (SVM) based binary classifier that classifies bug reports as missing or not missing expected behaviors (or steps to reproduce). S2RMiner, proposed by Zhao et al. [45] also apply machine learning techniques to bug reports. Different from DEMIBuD-ML, S2RMiner classifies sentences in bug reports as steps to reproduce (S2R) or non-S2R. Our approach is similar to DEMIBuD-ML and S2RMiner in that all of them apply machine learning based classification techniques to bug reports. However, our approach differs from DEMIBuD-ML and S2RMiner in that our approach classifies

action words whereas DEMIBuD-ML and S2RMiner classify either the whole bug report or sentences in bug reports.

## 6 CONCLUSIONS AND FUTURE WORK

Automated validation of bug report is highly desirable. However, existing approaches rely on manually constructed vocabulary and classification of action words, which prevents such automated approaches from achieving their maximal potential. To this end, in this paper, we propose an automated approach to identify and classify action words in mobile apps' bug reports. The key insight of the approach is that the contexts of the action words are critical for the classification of action words. Consequently, the proposed approach leverages natural language processing, semantic dependency trees, and static source code analysis to identify action words as well as their contexts, i.e., the enclosing segments, potential UI elements, and types of target elements. We also manually construct a training dataset for the training and evaluation of the proposed approach. Evaluation results suggest that the proposed approach is accurate. Evaluation results on publicly available datasets suggest that the proposed approach can significantly enhanced the state-of-the-art approaches in automated validation of bug reports.

As future work, we intend to expand the manually labelled training data. To build the data, we manually analyzed only 525 bug reports. In contrast, to construct vocabularies, Zhao et al. [46] manually analyzed 813 bug reports whereas Fazzini et al. [15] analyzed 400 tutorials on mobile apps. In future, if we can analyze more bug reports to expand the dataset, we may further improve the performance of the proposed approach. We also intend to extend the proposed approach to handle bug reports of traditional GUI applications (e.g., Windows applications).

## ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China under Grant No.: 61690205, 61772071

## REFERENCES

- [1] 2019. Bugzilla. <https://www.bugzilla.org/>.
- [2] 2019. Flutter's bug report, no.34330. <https://github.com/flutter/flutter/issues/34330>.
- [3] 2019. Github Issue Tracker. <https://github.com/issues>.
- [4] 2019. Google Code Issue Tracker. <https://code.google.com/archive/>.
- [5] 2019. LibreNews's bug report, no.22. <https://github.com/milesmcc/LibreNews-Android/issues/22>.
- [6] 2019. Lockwise's bug report, no.783. <https://github.com.mozilla-lockwise/lockwise-android/issues/783>.
- [7] 2019. MvvmCross's bug report, no.2532. <https://github.com/MvvmCross/MvvmCross/issues/2532>.
- [8] 2019. spaCy. <https://spacy.io/>.
- [9] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. IEEE, 133–143.
- [10] Dankmar Böhning. 1992. Multinomial Logistic Regression Algorithm. *Annals of the Institute of Statistical Mathematics* 44, 1 (1992), 197–200.
- [11] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the Quality of the Steps to Reproduce in Bug Reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 86–96.
- [12] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting*

- on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017.* ACM, 396–407.
- [13] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.
- [14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, Volume 1 (Long and Short Papers)*. 4171–4186.
- [15] Mattia Fazzini, Martin Prammer, Marcelo d’Amorim, and Alessandro Orso. 2018. Automatically Translating Bug Reports into Test Cases for Mobile Apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16–21, 2018.* ACM, 141–152.
- [16] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating Effective Test Cases for Self-driving Cars from Police Reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019.* ACM, 257–267.
- [17] Jianfeng Gao, Michel Galley, and Lihong Li. 2018. Neural Approaches to Conversational AI. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL ’18), Melbourne, Australia, July 15–20, 2018, Tutorial Abstracts*. 2–7.
- [18] Jerzy W. Grzymala-Busse. 1991. On the Unknown Attribute Values in Learning from Examples. In *Methodologies for Intelligent Systems, 6th International Symposium, ISMIS ’91, Charlotte, NC, USA, October 16–19, 1991, Proceedings*. 368–377.
- [19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.
- [20] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10–11, 2013.* IEEE, 15–24.
- [21] Liadh Kelly, Hanna Suominen, Lorraine Goeuriot, Mariana L. Neves, Evangelos Kanoulas, Dan Li, Leif Azzopardi, René Spijker, Guido Zuccon, Harrison Scells, and João R. M. Palotti. 2019. Overview of the CLEF eHealth Evaluation Lab 2019. In *Proceedings of the 10th International Conference of the CLEF Association, CLEF2019, Lugano, Switzerland, September 9–12*. 322–339.
- [22] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25–29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. 1746–1751.
- [23] Igor Kononenko. March 6–8, 1991. Semi-Naive Bayesian Classifier. In *European Working Session on Machine Learning (EWSL ’91), Porto, Portuga*. 206–219.
- [24] Andy Liaw, Matthew Wiener, et al. 2002. Classification and Regression by RandomForest. *R News* 2, 3 (2002), 18–22.
- [25] Xiao Luo and A. Nur Zincir-Heywood. 2005. Evaluation of Two Systems on Multi-class Multi-label Document Classification. In *Foundations of Intelligent Systems, 15th International Symposium, ISMIS 2005, Saratoga Springs, NY, USA, May 25–28, 2005, Proceedings*. Springer, 161–169.
- [26] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated Generation of Reproducible Test Cases for Android Apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, HotMobile 2019, Santa Cruz, CA, USA, February 27–28, 2019.* ACM, 99–104.
- [27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of 27th Annual Conference on Neural Information Processing Systems*. 3111–3119.
- [28] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing Bug Reports for Android Applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, Bergamo, Italy, August 30 – September 4*. 673–686.
- [29] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019.* IEEE Press, 188–199.
- [30] Sankar K. Pal and Sushmita Mitra. 1992. Multilayer Perceptron, Fuzzy Sets, and Classification. *IEEE Trans. Neural Networks* 3, 5 (1992), 683–697.
- [31] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P. Markatos, and Thomas Karagiannis. 2013. Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23–25, 2013.* ACM, 277–290.
- [32] David Martin Powers. 2011. Evaluation: from Precision, Recall and F-measure to ROC, Informedness, Markedness and Correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.
- [33] J. Ross Quinlan. 1989. Unknown Attribute Values in Induction. In *Proceedings of the Sixth International Workshop on Machine Learning (ML 1989), Cornell University, Ithaca, New York, USA, June 26–27, 1989*. 164–168.
- [34] Gunnar Rätsch, Takashi Onoda, and Klaus-Robert Müller. 1998. Regularizing AdaBoost. In *Advances in Neural Information Processing Systems (NIPS), Denver, Colorado, USA, November 30 – December 5*. 564–570.
- [35] Siva Reddy, Danqi Chen, and Christopher D. Manning. 2019. CoQA: A Conversational Question Answering Challenge. *Transactions of the Association for Computational Linguistics* 7 (2019), 249–266.
- [36] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15–19, 2014.* IEEE, 143.
- [37] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional Recurrent Neural Networks. *IEEE Trans. Signal Processing* 45, 11 (1997), 2673–2681.
- [38] scikit-Learn. 2019. API Specification on Weight of Categories. [https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LogisticRegression.html?highlight=class\\_weight](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=class_weight).
- [39] Minzhu Shen. 2020. Replication Package for Maca. <https://github.com/sakura182/Maca>.
- [40] Johan AK Suykens and Joos Vandewalle. 1999. Least Squares Support Vector Machine Classifiers. *Neural Processing Letters* 9, 3 (1999), 293–300.
- [41] Hongliang Yan, Yukang Ding, Peihua Li, Qilong Wang, Yong Xu, and Wangmeng Zuo. 2017. Mind the Class Weight Bias: Weighted Maximum Mean Discrepancy for Unsupervised Domain Adaptation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21–26, 2017*. 945–954.
- [42] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015.* IEEE, 658–668.
- [43] Tingting Yu, Tarannum S. Zaman, and Chao Wang. 2017. DESCRY: Reproducing System-level Concurrency Failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4–8, 2017.* ACM, 694–704.
- [44] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13–16, 2010.* ACM, 321–334.
- [45] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *Reuse in the Big Data Era - 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26–28, 2019, Proceedings*. Springer, 100–111.
- [46] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019.* IEEE Press, 128–139.



# Data Loss Detector: Automatically Revealing Data Loss Bugs in Android Apps

Oliviero Riganelli  
oliviero.riganelli@unimib.it  
University of Milano - Bicocca  
Milan, Italy

Simone Paolo Mottadelli  
s.mottadelli2@campus.unimib.it  
University of Milano - Bicocca  
Milan, Italy

Claudio Rota  
c.rota30@campus.unimib.it  
University of Milano - Bicocca  
Milan, Italy

Daniela Micucci  
daniela.micucci@unimib.it  
University of Milano - Bicocca  
Milan, Italy

Leonardo Mariani  
leonardo.mariani@unimib.it  
University of Milano - Bicocca  
Milan, Italy

## ABSTRACT

Android apps must work correctly even if their execution is interrupted by external events. For instance, an app must work properly even if a phone call is received, or after its layout is redrawn because the smartphone has been rotated. Since these events may require destroying, when the execution is interrupted, and recreating, when the execution is resumed, the foreground activity of the app, the only way to prevent the loss of state information is to save and restore it. This behavior must be explicitly implemented by app developers, who often miss to implement it properly, releasing apps affected by *data loss* problems, that is, apps that may lose state information when their execution is interrupted.

Although several techniques can be used to automatically generate test cases for Android apps, the obtained test cases seldom include the interactions and the checks necessary to exercise and reveal data loss faults. To address this problem, this paper presents *Data Loss Detector* (DLD), a test case generation technique that integrates an exploration strategy, data-loss-revealing actions, and two customized oracle strategies for the detection of data loss failures.

DLD revealed 75% of the faults in a benchmark of 54 Android app releases affected by 110 known data loss faults, and also revealed unknown data loss problems, outperforming competing approaches.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Android, data loss, test case generation, validation, mobile apps.

### ACM Reference Format:

Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data Loss Detector: Automatically Revealing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397379>

Data Loss Bugs in Android Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397379>

## 1 INTRODUCTION

In the last decade, mobile apps have increasingly gained importance and popularity. Recent studies revealed that people spend more than 3h per day on their smartphones on average [23] and that 90% of this time is typically devoted to the use of mobile apps [21]. Indeed, people use mobile apps to perform a huge variety of tasks, including reading e-mails, listening to music, making orders and payments, and playing games.

Among the available ecosystems for the distribution of mobile apps, the Android ecosystem is the largest and most used one: its market share is almost 75% [29] and its official store, the Google Play Store, includes almost 3.0 millions of apps [30].

Android apps consist of components, such as activities, fragments, and services, whose behavior must comply with well-defined lifecycles [9, 11, 13]. For instance, activities can be in states such as created, paused, resumed, and stopped, and transitions between these states produce *callbacks* that must be handled by the activities.

Interestingly, some of these callbacks might be particularly tricky to implement. This is the case of the callbacks produced by *stop-start* events, which are system events that may force the destruction and then the (re-)instantiation of a running activity. Stop-start events occur every time the execution of an app is stopped and then resumed. Typical cases include answering a phone call, switching between apps, and rotating the smartphone to change its layout.

When a stop-start event occurs, the difficult task for the app is to handle the destruction of the current activity in a way that is later possible to resume the execution at the same point it was interrupted. This is done by saving the values of all the relevant state variables before the activity is destroyed, and retrieving these values when the execution is resumed. With the exception of some specific cases (e.g., the widgets with a non-empty *android:id* property), it is a responsibility of the developer to implement this behavior. In particular, developers have to implement both the logic necessary to save the state of an activity in the *onSaveInstanceState()* callback method and the logic to resume its state in the *onRestoreInstanceState()* callback method [10]. Unfortunately, this implementation might be wrong and might introduce misbehaviors in the apps [16].

When a stop-start event is not properly handled, the Android app is said to be affected by a *data loss* fault, that is, a fault that causes one or more state variables to lose their values. Data loss faults may affect the correctness of the apps in many different ways. In the best cases, they force the users to enter again inputs that had already been entered, deteriorating the quality of the user experience. In the worst cases, they generate activities with an inconsistent state, which causes the apps to produce wrong outputs or even crashes.

Data loss faults can be extremely pervasive. Adamsen *et al.* [1] considered the execution of system events jointly with test suites and reported that all the four apps used in their evaluation were affected by data loss faults. Riganelli *et al.* [26] analyzed 428 Android apps and found that at least 82 (19.6%) of the apps were affected by data loss faults. Finally, Amalfitano *et al.* [2] studied 68 open source apps reporting data loss faults in 60 of them (88.2%).

Test case generation techniques could be potentially used to reveal data loss faults. Indeed, a number of automatic test case generation techniques are available for Android apps. For instance, Monkey can generate test cases randomly [12], A<sup>3</sup>E can generate tests systematically using a depth-first strategy [3], DroidBot [19] and Stoat [31] exploit a model-based approach, and Sapienz uses evolutionary algorithms [20]. While these approaches are able to reveal several interesting faults, they are *ineffective* against data loss problems for two reasons: (i) they do not include operations that cause stop-start events, and (ii) they are not equipped with oracles strong enough to detect non-crashing data loss failures.

Some techniques have been designed to extend the fault discovery capability of existing test suites to data loss problems. For instance, Thor systematically injects neutral event sequences, including stop-start events, into existing test cases to augment their failure detection capability [1]. Quantum behaves similarly but it starts from a GUI model of the app [33]. Although useful, their applicability is limited to apps equipped with comprehensive test suites or GUI models. ALARic randomly generates test cases that include stop-start events and detects data loss problems by comparing the state of the app under test before and after a stop-start event is generated [24]. ALARic can successfully reveal data loss faults, but the adopted exploration strategy and oracles have limited effectiveness, as reported in our evaluation.

In this paper, we present *Data Loss Detector* (DLD), an automatic testing technique that can reveal data loss problems in Android apps. DLD integrates three capabilities to effectively reveal data loss problems: (i) a *biased model-based* exploration strategy that steers the exploration towards (new) app states that may be affected by data loss problems, (ii) *data-loss-revealing actions* that increase the likelihood to expose data loss problems, and (iii) two state-based oracles, a *snapshot-based* oracle and a *property-based* oracle, that have a high data loss detection accuracy, especially if used jointly.

Compared to Thor [1] and Quantum [33], DLD does not require a pre-existing test suite or a pre-existing GUI model, neither requires an initial ripping phase, but iteratively and continuously generates test cases according to the allocated time budget. Finally, the biased model-based exploration and the data-loss-revealing actions exploited in DLD allow a more effective data loss detection than the strategy implemented in Alaric [24].

We empirically evaluated DLD using the benchmark by Riganelli *et al.* [26], which includes 110 data loss problems affecting 54 app

releases. DLD automatically detected 83 of the 110 (75%) data loss problems. DLD also revealed 35 data loss faults that were not part of the benchmark, but were reported online in bug reports, and 232 previously unknown data loss faults. Overall DLD revealed three times the data loss faults revealed by competing approaches. We finally submitted the data loss faults that still affect apps nowadays online to app developers who positively reacted to our bug reports.

In a nutshell, this paper makes the following contributions.

- It describes an exploration strategy, data-loss-revealing actions, and two state-based oracles that can be incorporated in testing techniques to reveal data loss problems,
- It delivers the Data Loss Detector (DLD) technique, which is implemented as an extension of the DroidBot [19] test case generation tool for Android,
- It reports the largest empirical evaluation about data loss detection available so far, considering hundreds of data loss faults,
- It delivers the tool and the experimental material freely available online at the following url: <https://bit.ly/30XLygW>

This paper is organized as follows. Section 2 discusses data loss faults and exemplifies typical failures that can be experienced with Android apps. Section 3 describes the main technical contributions of this paper, including the biased model-based exploration strategy, the data-loss-revealing actions, and the automatic oracles. Section 4 reports empirical results. Section 5 discusses related work. Finally, Section 6 provides final remarks.

## 2 DATA LOSS FAULTS

In this section we describe data loss faults and the Android components that can be affected by these faults: activities and fragments.

An Android *activity* is a component that implements a screen of an app and the logic to handle that screen. To partition a screen into smaller units, activities can contain a number of *fragments*, each one containing both some graphical elements and the logic to handle them. Both activities and fragments have their own lifecycle [9, 13].

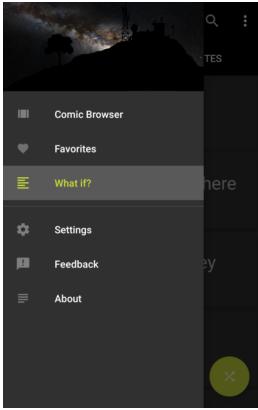
Specific sequences of system events may have a direct impact on the lifecycle of activities and fragments.

**Definition 2.1.** A *stop-start* event is a sequence of system events that may cause a running activity (or fragment) to be destroyed and then recreated<sup>1</sup>.

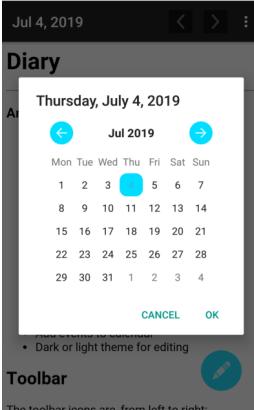
Stop-start events are generated when the execution of an activity must be temporarily suspended. There are many common situations that produce stop-start events. For instance, answering a phone call requires suspending the execution of the app, and thus also of the current activity, until the call ends. Moving an app to the background may cause its activities to be destroyed. When the app is moved again to the foreground, the status of the foreground activity has to be recreated. The rotation of the screen finally causes the destruction of the current activity that must be redrawn with a new layout.

Note that all these situations are neutral from the user's perspective, that is, they are not expected to change the status of the app: users expect to find the status of an app unchanged after they have answered a phone call, after the app has been moved to the background and then to the foreground, and after the screen has

<sup>1</sup>In the rest of the paper we refer only to activities for simplicity, but all the concepts apply to both activities and fragments.



(a) Application crash in EASY XKCD v6.0.4



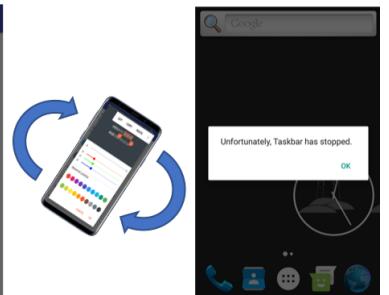
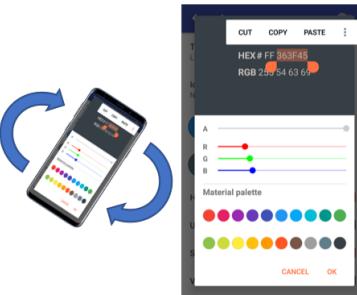
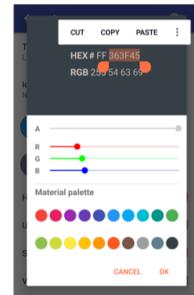
(b) Disappearance of a Dialog in DIARY v1.26



(c) Appearance of a Dialog in CYCLESTREETS v3.5



(d) Change of EditText values in BEECOUNT v2.7.4



(e) Loss of the internal state in TASKBAR v3.0.3

Figure 1: Examples of data loss failures after a stop-start event.

been rotated. However, this behavior is not provided for free by Android, but it must be guaranteed by developers who have to implement the logic to save and retrieve the status of the activities. If this piece of logic is not implemented correctly, stop-start events are no longer neutral and state information might be lost, causing a *data loss* problem.

**Definition 2.2.** A *data loss* problem occurs when data is accidentally deleted or state variables are accidentally assigned with default or initial values.

Data loss faults can be the source of diverse failures [18]. Indeed, the initialization of some program variables with wrong values (e.g.,

to the default value) can be the cause of unpredictable behaviors. Based on our evaluation, we isolated five main failure patterns which are exemplified in Figure 1:

- *crashes*: the app may simply crash. This is exemplified in Figure 1 (a) where the EASY XKCD app crashes after a rotation of the screen.
- *destroyed GUI elements*: some graphical elements may disappear forcing the user to repeat operations. This is exemplified in Figure 1 (b) where the calendar dialog in the DIARY app disappears after a rotation of the screen.

- *phantom GUI elements*: some graphical elements may erroneously appear, forcing the user to perform unwanted and unclear interactions. This is exemplified in Figure 1 (c) where a dialog appears in the CYCLESTREETS app after a rotation of the screen.
- *modified values*: some elements may unexpectedly change their values, resulting in misbehaviors of the app. This is exemplified in Figure 1 (d) where multiple text fields change to 0 in the BEECOUNT app after a rotation of the screen.
- *compromised internal state*: the internal state of the app might be compromised causing visible misbehaviors in the interactions that follow the activation of the data loss. This is exemplified in Figure 1 (e). The initial rotations of the screen compromise the status of the TASKBAR app without producing any visible misbehavior, until the last rotation causes a crash.

### 3 DATA LOSS DETECTOR

Data Loss Detector addresses data loss faults combining three key ingredients: (i) a biased model-based exploration strategy, which increases the likelihood to explore states that may expose data loss faults; (ii) data-loss-revealing actions, which interact with the app under test stimulating behaviors that are prone to data loss, and (iii) data loss oracles, which analyze the behavior of the app under test to detect data loss failures.

#### 3.1 Biased Model-Based Exploration

The test case generation strategy implemented in DLD consists in visiting as many states as possible and incrementally testing the newly discovered states to detect data loss faults. To this end, the strategy builds a GUI model that represents the visited states and the executed actions. The model serves two main purposes: to distinguish the already visited (and tested) states from the new ones, and to bias the exploration towards the execution of actions that may potentially lead to states never visited before.

**Definition 3.1.** A *GUI model* is a non-deterministic finite state automaton  $(Q, \Sigma, q_0, \delta)$ , where  $Q$  is a finite set of *abstract states*;  $\Sigma$  is the finite set of *events* that can be triggered from such abstract states, such as clicks, swipes, or *stop-start* events;  $q_0 \in Q$  is the *initial abstract state*;  $\delta : Q \times \Sigma \rightarrow \wp(Q)$  is the *transition function*, which, given  $q \in Q$  and  $e \in \Sigma$ , returns the set of *abstract states* reachable from  $q$  by executing  $e$ .

To effectively test an app, it is important to represent states at an appropriate abstraction level. A too abstract representation would collapse many concrete states into a single abstract state causing several relevant states not being tested for data loss. A too concrete representation would cause an enormous waste of time, testing for data loss states with irrelevant differences. The abstraction level used by DLD derives from the following two observations.

- *Concrete values do not matter*: a GUI state representation might include all the widgets with their properties and values. As a consequence, two concrete states that differ for a single value assigned to a label or to an input field would produce different abstract states that would be tested for data loss. This is a waste of resources, because as long as some values are assigned to the various GUI elements, the data loss is likely to show up regardless of the concrete values assigned to these elements.

- *Totally ignoring the content of screens is too inaccurate*: a GUI state representation might be so abstract to only consider the name of the current Android activity as identifier of the abstract state, that is, the number of abstract states matches with the number of Android activities implemented in the tested app. This strategy totally ignores the content of the screens and is likely to miss all those data loss faults that can be exercised only in a specific state of an activity.

Based on these two observations, DLD uses an abstraction that ignores the concrete values, but still discriminates the relevant distinct states associated with a same Android activity. To this end, it uses the *enabledness abstraction*, which has already been used in other contexts to distinguish the states of software applications [7, 8], and preliminary experienced in Quantum to reveal GUI interaction faults [33]. In this case, the idea is to distinguish states not based on the content of the screen, but based on the set of actions (i.e., events) that are allowed. Intuitively, it is worth distinguishing states that enable a different set of behaviors for the software. For instance, two different values in an input field produce two different abstract states only if one of the two values enables operations that are otherwise disabled.

**Definition 3.2.** An *abstract state*  $q \in Q$  associated with a concrete state  $s$  is a pair  $(a, E)$ , where  $a$  is the name of the current activity in  $s$  and  $E \subseteq \Sigma$  is the set of events enabled in  $s$ .

The test case generation strategy is biased towards the execution of new actions that may lead to new (abstract) states potentially affected by data loss. In particular, every time an action is executed, DLD has a probability  $\epsilon$  to choose an event at random, and a probability  $1 - \epsilon$  to choose an event that has never been executed in the current abstract state based on the available GUI model.

DLD can generate five types of actions during the exploration (four actions inherited from DroidBot plus a scroll action added to reach every view in a window), in addition to the data-loss-revealing actions described in next section:

- *TouchEvent*, which executes a tap on a clickable view;
- *LongTouchEvent*, which executes a long tap on a clickable view;
- *SetTextEvent*, which writes text inside an editable view;
- *KeyEvent*, which presses a navigation button (e.g. “Home”);
- *ScrollEvent*, which executes a swipe on a scrollable view;

DLD incrementally updates the GUI model after the execution of every event. The testing activity stops after a budget that can be expressed as a number of actions to be performed or as an amount of time to be allocated for testing.

#### 3.2 Data-Loss-Revealing Actions

The exploration activity described in Section 3.1 is combined with the execution of data-loss-revealing actions that have the objective to reveal data loss problems, if present.

DLD includes two data-loss-revealing actions: one is executed *systematically every time a new abstract state is reached* (*systematic data-loss-revealing action*), while the other is executed *probabilistically at every abstract state* (*probabilistic data-loss-revealing action*).

The systematic data-loss-revealing action is composed of the following sequence of steps:

- (1) *fill-in*: DLD interacts with all the input views (e.g., text field, combo box, check box) entering non-empty values different from the default values,
- (2) *save state*: the current GUI state is saved to later check if any data loss occurred. The way the state is saved depends on the oracle strategy adopted (see Section 3.3),
- (3) *double screen rotation*: the screen rotation is a stop-start event. It is executed twice to reach a state that should be exactly the same that was saved, if no data loss occurred,
- (4) *check state*: the current GUI state is compared to the saved state to determine if any data loss occurred. The way the comparison is performed depends on the oracle strategy (see Section 3.3),
- (5) *scroll down*: some Android activities may include visual elements that span over the size of the screen. To make sure the reached abstract state is fully tested for data loss, including the elements that might be outside the screen, if the *check state* step has not revealed a data loss, DLD executes a scroll down action that may make new elements appear. If this happens, a new abstract state might be reached, which would be again systematically tested for data loss.

The systematic data-loss-revealing action already guarantees an accurate validation of the state space, as defined by our abstraction strategy. However, there might be certain data loss faults that depend on internal state information that does not produce any difference in the GUI state of the app. For instance, the window for setting a timer in the ANTENNAPOD app generates a data loss only if a podcast has been loaded before from another window. The fact that a podcast was loaded does not result in any visible difference in the timer window, and thus the abstraction strategy cannot capture the difference between the state that exposes the data loss fault and the one that does not. To address these cases, when an already visited state is encountered, DLD enables the probabilistic data-loss-revealing action that has the same probability of the other actions to be selected. The probabilistic data-loss-revealing action performs the sequence of events from step 2 to 4 of the systematic data-loss-revealing action.

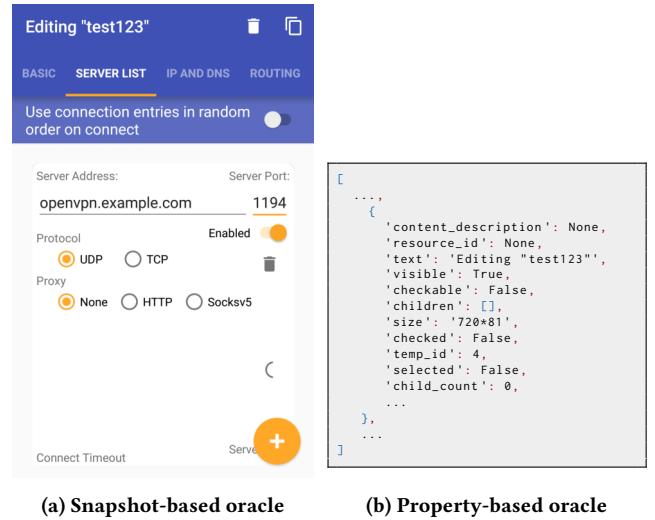
In a nutshell, the exploration works as follows. When a new abstract state is encountered, DLD executes the systematic data-loss-revealing action. Otherwise, DLD identifies the set of actions  $A$  that can be executed on the current GUI state based on the five types of supported actions (see Section 3.1). Namely a subset of them,  $A^+$ , has already been executed based on the GUI model, while the others,  $A^-$ , have not been executed yet. DLD executes with probability  $\epsilon$  a random action, that is, an action in the set  $A \cup \{\text{probabilistic data-loss-revealing action}\}$ , and with probability  $1 - \epsilon$  an action that has not been executed yet, that is, an action in  $A^- \cup \{\text{probabilistic data-loss-revealing action}\}$ .

### 3.3 Data Loss Oracles

Data loss problems do not always cause crashes. On the contrary, apps can present a range of misbehaviors as discussed in Section 2. DLD uses oracles based on the fact that the operations that exercise the data loss faults are expected to be neutral, thus leaving the status of the app unchanged. As anticipated in Section 3.2, the basic

DLD strategy is to collect the GUI state of the app before and after the execution of actions that may have triggered a data loss failure and compare the states to detect it.

DLD defines two oracle strategies, which can be used either independently or jointly: the snapshot-based oracle and the property-based oracle. The *snapshot-based oracle* takes a screenshot of the app before and after a data loss might have occurred and compares the images to detect failures. The *property-based oracle* analyzes the GUI state and collects all the views and all their properties, and compares these two sets of properties to detect failures. Figure 2 shows the state information captured by the snapshot-based and the property-based oracles for a same GUI state of the OPENVPN app. The former oracle stores a screenshot of the app, as shown in Figure 2 (a), while the latter oracle stores the properties of the views in Python dictionary format, as shown in Figure 2 (b).



**Figure 2: The information saved by the two types of oracles for a same state of OPENVPN.**

More rigorously, the two oracle strategies collect and compare state information as follows.

#### Snapshot-based oracle

*State Information*: DLD first takes a screenshot of the device. The recorded image is then converted into a grayscale image, which is faster to compare than a colourful image. Finally, DLD crops the header and the footer of the image because it contains information that changes over time regardless of data loss, such as the current time and the battery level. The resulting image is the retrieved representation of the current state.

*State Comparison*: DLD compares the two states by comparing the two corresponding images pixel by pixel. Since a blinking cursor might cause a small level of noise in the representation of the images, the comparison fails only if more than 15 pixels every 10,000 pixels are different.

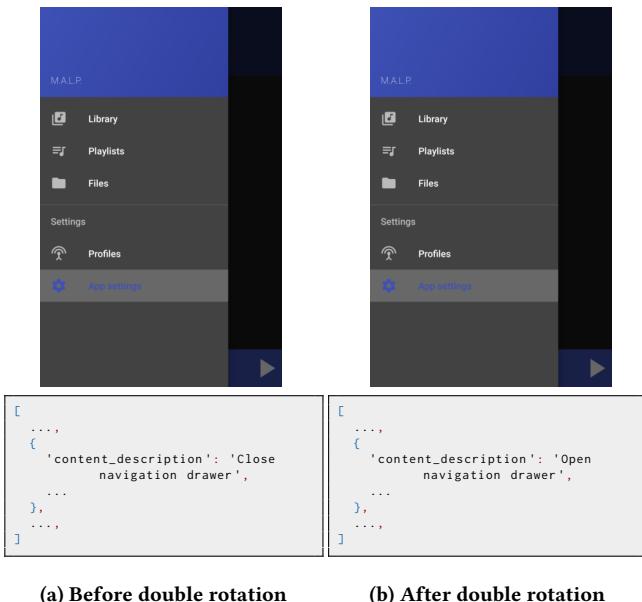
#### Property-based oracle

*State Information*: DLD retrieves all the views, including their properties and their hierarchical organization. The retrieved values are represented in a Python dictionary format.

**State Comparison:** DLD compares the two states by comparing their Python-based representation. The comparison fails if one of the attribute values is different or the hierarchical structure of the views is not preserved.

Note that although the two strategies may seem redundant, they are not, as confirmed by our experiments. In particular, there are a number of data loss problems that can only be detected by one strategy. For instance, Figure 3 shows the case of a data loss failure that has been detected by the property-based oracle only. In fact, the two snapshots of the MALP app are visually identical, but actually the content descriptor has changed its value. On the other hand, Figure 4 shows the case of a data loss failure that has only been detected by the snapshot-based oracle. The set of properties, not shown in the figure, are exactly the same for the two states, but the app has lost the zoom, as clearly visible from the snapshots.

Interestingly, the two oracles can be combined, so that a failure is reported if just one of the two oracle strategies reports a failure.



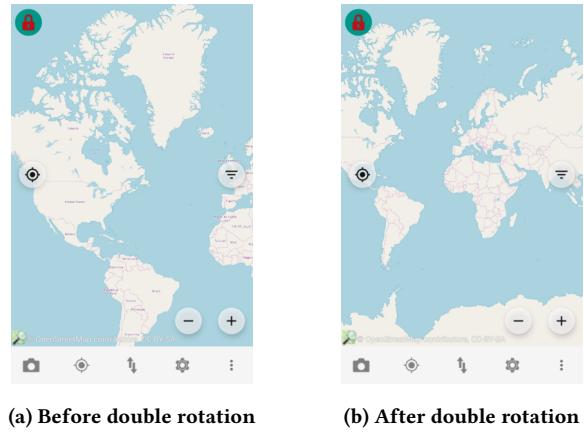
**Figure 3: A data loss failure detected by the property-based oracle only in MALP 3d31062.**

## 4 EVALUATION

This section presents the empirical results obtained by experimenting DLD with a benchmark of 110 data loss faults, in comparison to the ALARic [24] and Quantum [33] test case generation techniques. We first describe our implementation of DLD, we then introduce the research questions and the subject applications. We finally present the results obtained for each research question in details, and discuss threats to validity.

### 4.1 Implementation

We implemented DLD as an extension of *DroidBot* [19], which is a state-of-the-art test case generator for Android that does not



**Figure 4: A data loss failure detected by the snapshot-based oracle only in VESPUCCI OSM EDITOR v10.2.**

implement features for the detection of data loss faults. DLD inherits from DroidBot the capability to perform a specific sequence of actions before testing a target app. This feature can be used to setup the initial state of the app under test. In our evaluation, we used this capability to: authenticate into the apps that require a log in, setup an initial project in MGit, and grant permissions in QUICKLYRIC.

As outcome of the testing process, DLD generates both a report with the revealed data loss faults and reproducible test cases. Each data loss is described in terms of the screenshots and the set of GUI properties collected before and after the data loss is observed. DLD is available at <https://bit.ly/30XLyGw>.

### 4.2 Research Questions

We evaluate DLD by studying the following five research questions.

- *RQ0 - What is the  $\epsilon$  that provides the best exploration?* This research question studies the impact of the  $\epsilon$  parameter on the effectiveness of the exploration. The result of this research question is used to configure DLD to address the other research questions.
- *RQ1 - How effective is DLD with data loss problems?* This research question investigates the effectiveness of DLD considering two perspectives captured by the following sub-RQs.
  - *RQ1.1 - What is the data loss discovery capability of DLD?*
  - *RQ1.2 - What is the rate of the spurious oracle violations reported by DLD?*
- *RQ2 - Is DLD more effective than state-of-the-art techniques?* This research question is decomposed into the following sub-RQs.
  - *RQ2.1 - What is the relative effectiveness of DLD, ALARic, and Quantum?* This sub-RQ compares DLD to ALARic and Quantum.
  - *RQ2.2 - What are the main factors that determine the effectiveness of DLD?* This sub-RQ investigates the factors that allow DLD to reveal more data loss faults than competing techniques.
- *RQ3 - What is the tradeoff between the snapshot- and property-based oracles?* This research question investigates the tradeoff between the two oracle strategies, measuring the data loss faults revealed by the snapshot-based oracle only, by the property-based oracle only, and by both.

- *RQ4 - Are data loss faults relevant to developers?* This research question studies how app developers react to the presence of data loss problems in their apps.

### 4.3 Subject Applications

In our empirical evaluation we used the benchmark by Riganelli *et al.* [26], which includes 110 data loss problems affecting a total of 48 Android apps and 54 app releases. Each data loss fault is equipped with an Appium test case [14] that can be executed to reproduce the problem. All the experiments have been conducted with the Genymotion v3.0.2 Android emulator using an emulated Google Nexus 5 device equipped with Android 6.0 API 23 and 2 GB of RAM. In the evaluation, we followed the practice of other studies [32] performing three runs of 3 hours each per tested app for both DLD and ALARic, for a total of 42 days of uninterrupted computation.

### 4.4 RQ0 - What is the $\epsilon$ that provides the best exploration?

This research question investigates the impact of the  $\epsilon$  parameter on the effectiveness of the approach. To perform this initial study we selected 3 apps from the benchmark. To cover the range of situations that might be faced with the full benchmark, we selected the apps with the smallest, medium, and highest number of activities, which are EQUATE (2 activities), CALENDAR NOTIFICATION (13 activities), and TWIDERE (52 activities), respectively.

To assess the capability to explore the app and potentially reveal data loss problems we measured *activity coverage*, which is the percentage of activities covered in a test session. We started with  $\epsilon = 0$ , which consists of a strategy that always privileges the execution of actions that have not been executed before, based on the incrementally constructed GUI model. We then increased the  $\epsilon$  parameter by 0.1 to study its impact on the results. We stopped with  $\epsilon = 0.2$ , which produced a significant decrease on the effectiveness of the exploration. Results are summarized in Table 1. We finally selected  $\epsilon = 0.1$  to address the other research questions.

**Table 1: Impact of the  $\epsilon$  parameter on activity coverage.**

$\epsilon$	Activity Coverage
$\epsilon = 0$ (always new actions)	52%
$\epsilon = 0.1$ (random actions with probability 0.1)	60%
$\epsilon = 0.2$ (random actions with probability 0.2)	44%

### 4.5 RQ1 - How effective is DLD with data loss problems?

This research question investigates the capability of DLD to reveal the data loss problems in the benchmark. To answer this research question, we manually analyzed every report produced by DLD to distinguish the actual data loss problems from the irrelevant spurious violations. In particular, we classified a reported data loss as spurious if one of the two following conditions holds: (i) the state of the app after the double screen rotation is taken too early, while the activity is still recreating, making the oracle to fail its check or (ii) the difference reported by the oracle cannot be considered a data loss (e.g., because the tested app shows the current time which

obviously changes after the screen has been rotated twice). In the vast majority of the cases the reports were enough to classify data loss problems. We reproduced the problem in the unclear cases. For this research question, we detected data loss problems using both the snapshot-based and the property-based oracles. We analyze their relative fault detection ability with RQ3.

We checked each data loss problem revealed by DLD, distinguishing if it is a *benchmark data loss*, that is, a problem that is part of the benchmark we used; an *online data loss*, that is, a problem already reported online that is not part of the benchmark (for all the apps in the benchmark we searched online for additional data loss faults, and we used the snapshots, the activity name and the fields reported to lose their values to determine if a discovered data loss matches with the online data loss), or a *new data loss*, that is, an unknown data loss problem (to the best of our ability to search for reported problems). We refer to the union of the benchmark and online data loss faults as the *known data loss faults*.

We report the *number of activities affected by a data loss problem* found by DLD. They intuitively correspond to different faults and different fixes to be implemented in different activities. The only exception is with the known data loss faults. Since some of the data loss faults in the benchmark affect the same activity, we actually report the precise number of faults in the benchmark that have been revealed. Finally, when only spurious violations are detected for an activity, we report it as a *spurious data loss*.

Table 2 column DLD (left part of the table) shows the results that DLD obtained for all the app releases considered in the study. Since every row represents the outcome of three runs, when applicable, we report both average values and total values. Column # Activities indicates the total number of activities in each app. Column Activity Coverage avg (total) reports the activity coverage achieved in average and in total in the three executions. Column Activities with Data Loss indicates the number of activities affected by at least a data loss revealed by DLD. Column Benchmark Data Loss avg (total)/existing indicates the average and total number of data loss faults that have been revealed by DLD out of the ones present in the benchmark. For example in the BookCATALOGUE app, DLD revealed 5 data loss faults of the benchmark on average, achieved a total of 6 data loss faults revealed across the three runs, out of a total of 7 data loss faults present in the benchmark. Similarly, column Online Data Loss avg (total)/existing indicates the average and total number of data loss faults reported online revealed by DLD. Column New Data Loss avg (total) indicates the average and total number of previously unknown data loss faults revealed. Column Spurious Data Loss avg (total) indicates the average and total number of activities that originated spurious violations only. Column Crashes reports the total number of activities that crashed due to data loss faults. The top part of the table lists the apps where no initial setup has been necessary, while the bottom part of the table lists the apps that have been addressed as discussed in Section 4.1.

**4.5.1 RQ1.1 - What is the data loss discovery capability of DLD?** In terms of exploration, DLD managed to visit 66% of the activities, revealing 298 activities affected by data loss faults (41% of the total number of activities). It detected 83 of the 110 faults in the benchmark (75%), 35 out of the 58 (60%) additional data loss faults that we found online, and revealed 232 new data loss faults (an average of

Table 2: Results for DLD, and comparison to ALARic.

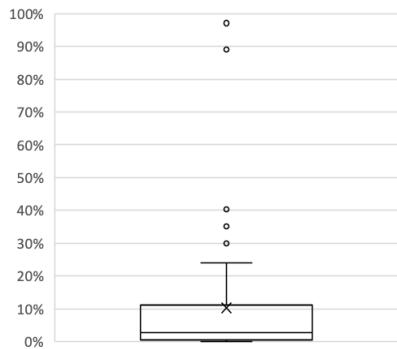
App name	Release	# Activities	DLD						ALARic					
			Activity Coverage avg (total)	Activities with Data Loss avg (total)	Benchmark Data Loss avg (total/existing)	Online Data Loss avg (total/existing)	New Data Loss avg (total)	Spurious Data Loss avg (total)	Crashes	Activities with Data Loss avg (total)	Benchmark Data Loss avg (total/existing)	Online Data Loss avg (total/existing)	New Data Loss avg (total)	Spurious Data Loss avg (total)
Amaze File Manager	v3.1.0-beta.1	4	100% (100%)	3	3 (3)/5	2 (2)/2	1(1)	0 (0)	1	1	1 (1)/5	0 (0)/2	0 (0)	0 (0)
AntennaPod	v1.5.2.0	16	33% (44%)	5	5 (5)/7	2 (2)/11	3 (3)	1 (1)	1	1	0 (0)/7	1 (1)/11	0 (0)	0 (0)
BeeCount	v2.4.7	8	96% (100%)	7	1 (1)/3	1 (1)/5	5 (5)	1 (1)	0	5	0 (0)/3	0 (0)/5	5 (5)	1 (1)
BookCatalogue	v5.2.0-RC3a	35	66% (71%)	21	5 (6)/7	-	12 (15)	0 (0)	1	7	2 (2)/7	-	4 (5)	6 (7)
Calendar Notification	v3.14.159	13	67% (69%)	8	3 (3)/3	1 (1)/1	4 (5)	0 (0)	1	2	1 (1)/3	0 (0)/1	1 (1)	0 (0)
CycleStreets	v3.5	11	55% (55%)	6	1 (1)/1	-	5 (5)	0 (0)	0	1	1 (1)/1	-	0 (0)	0 (0)
Diary	v1.26	3	100% (100%)	3	1 (2)/2	-	2 (2)	0 (0)	0	2	1 (1)/2	-	1 (1)	0 (0)
DNS66	v0.3.3	5	100% (100%)	3	1 (1)/1	-	2 (2)	0 (0)	0	1	0 (0)/1	-	1 (1)	2 (2)
Document Viewer	v2.7.9	9	48% (56%)	2	1 (1)/1	-	2 (2)	1 (1)	0	1	1 (1)/1	-	0 (0)	4 (4)
Easy xkcd	v6.0.4	9	74% (78%)	4	1 (1)/1	-	3 (3)	0 (0)	3	1	0 (0)/1	-	1 (1)	1 (1)
Equate	v1.6	2	100% (100%)	2	2 (2)/2	1 (1)/1	1 (1)	0 (0)	1	1	2 (2)/2	1 (1)/1	0 (0)	0 (0)
Etar Calendar	v1.0.10	12	42% (42%)	3	4 (4)/5	2 (3)/5	1 (1)	0 (0)	0	0	0 (0)/5	0 (0)/5	0 (0)	0 (0)
Firefox Focus	v4.0	6	44% (50%)	3	0 (0)/1	-	3 (3)	0 (0)	0	2	0 (0)/1	-	2 (2)	0 (0)
Flynn	v1.3.4	6	83% (83%)	4	0 (0)/1	-	4 (4)	0 (0)	0	3	0 (0)/1	-	3 (3)	0 (0)
Gadgetbridge	v0.25.1	20	28% (30%)	4	1 (1)/1	-	2 (3)	2 (2)	2	0	0 (0)/1	-	0 (0)	0 (0)
KISS Launcher	v2.25.0	2	100% (100%)	1	1 (1)/1	-	0 (0)	1 (1)	0	1	0 (0)/1	-	1 (1)	0 (0)
Loop Habit Tracker	v1.6.2	7	71% (71%)	4	2 (2)/6	-	1 (2)	1 (1)	0	2	0 (0)/6	-	2 (2)	2 (2)
MALP	3d31062	2	100% (100%)	1	1 (1)/1	-	0 (0)	0 (0)	1	0	0 (0)/1	-	0 (0)	0 (0)
MALP	v1.1.0	4	33% (50%)	2	2 (3)/4	-	1 (1)	0 (0)	1	1	0 (0)/4	-	1 (1)	0 (0)
MTG Familiar	v3.5.5	2	50% (50%)	1	1 (1)/1	-	0 (0)	0 (0)	0	1	0 (0)/1	-	1 (1)	0 (0)
NotePad	v2.3	3	67% (67%)	2	1 (1)/1	-	1 (1)	0 (0)	0	1	1 (1)/1	-	0 (0)	0 (0)
Omni Notes	v5.4.3	17	29% (35%)	4	0 (0)/1	-	4 (4)	0 (0)	0	1	0 (0)/1	-	1 (1)	1 (1)
OpenTasks	v1.1.13	9	78% (78%)	7	1 (1)/1	-	6 (6)	0 (0)	0	3	0 (0)/1	-	3 (3)	1 (1)
OpenVPN for Android	v0.7.5	13	46% (46%)	5	1 (1)/1	-	4 (4)	0 (0)	1	4	0 (0)/1	-	3 (4)	1 (1)
PassAndroid	v3.3.3	14	36% (36%)	4	2 (3)/3	8 (8)/8	1 (1)	0 (0)	1	3	1 (1)/3	4 (4)/8	1 (1)	1 (1)
Periodic Table	v1.1.1	3	100% (100%)	3	2 (2)/2	-	1 (1)	0 (0)	0	0	0 (0)/2	-	0 (0)	2 (2)
Port Knocker	v1.0.8	6	50% (50%)	2	2 (2)/3	0 (0)/1	0 (0)	0 (0)	0	2	0 (0)/3	0 (0)/1	2 (2)	1 (1)
Prayer Times	v3.6.6	22	32% (36%)	8	3 (6)/7	1 (1)/2	4 (5)	0 (0)	1	5	1 (1)/7	0 (0)/2	3 (4)	0 (0)
QuasselDroid	v0.11.5	5	40% (40%)	2	1 (1)/1	-	1 (1)	0 (0)	1	0	0 (0)/1	-	0 (0)	1 (1)
Simple Draw	v3.1.5	7	86% (86%)	3	0 (0)/1	-	3 (3)	0 (0)	0	0	0 (0)/1	-	0 (0)	1 (1)
Simple File Manager	v2.6.0	8	88% (88%)	5	1 (1)/1	-	3 (4)	0 (0)	2	1	1 (1)/1	-	0 (0)	1 (1)
Simple File Manager	v3.2.0	8	75% (75%)	5	1 (1)/1	-	3 (4)	0 (0)	1	1	1 (1)/1	-	0 (0)	0 (0)
Simple Gallery	v1.50	11	48% (55%)	5	1 (2)/4	1 (1)/7	3 (3)	0 (0)	0	1	1 (1)/4	0 (0)/7	0 (0)	1 (1)
Simple Solitaire	v2.0.1	7	93% (100%)	2	1 (1)/1	-	1 (1)	0 (0)	1	1	0 (0)/1	-	1 (1)	2 (2)
Simpletask	v10.0.7	11	67% (73%)	7	1 (1)/1	-	6 (6)	0 (0)	1	4	1 (1)/1	-	3 (3)	0 (0)
Syncthing	v0.9.5	9	93% (100%)	8	3 (4)/5	2 (2)/2	4 (5)	1 (1)	2	2	1 (1)/5	1 (1)/2	0 (0)	0 (0)
Taskbar	v3.0.3	21	26% (29%)	3	2 (2)/2	12 (13)/13	2 (2)	1 (1)	1	1	1 (1)/2	1 (1)/13	0 (0)	0 (0)
Tasks Astrid To-Do List Clone	v6.0.6	45	19% (27%)	10	0 (0)/1	-	7 (10)	1 (1)	2	1	0 (0)/1	-	1 (1)	0 (0)
Vespucci Osm Editor	v10.2	19	42% (47%)	7	1 (1)/1	-	5 (6)	1 (1)	0	5	1 (1)/1	-	4 (4)	0 (0)
Ville Checker	v4.4.0	6	67% (67%)	3	1 (1)/1	-	2 (2)	0 (0)	0	1	0 (0)/1	-	1 (1)	0 (0)
WiFi Analyzer	v1.9.2	4	75% (75%)	3	3 (3)/3	-	1 (1)	0 (0)	0	0	0 (0)/3	-	0 (0)	0 (0)
World Clock & Weather	v1.8.6	4	100% (100%)	4	0 (0)/1	-	4 (4)	0 (0)	0	1	0 (0)/1	-	1 (1)	1 (1)
TOTAL (all apps)		731	62% (66%)	298	83/110	35/58	232	0.26 (14)	38	71	19/97	8/58	50	0.74 (31)

4.3 new data loss faults revealed per app). In total, DLD revealed 350 data loss problems in 54 app releases, demonstrating a significant capability to detect data loss problems. Note that we started the empirical investigation knowing that less than 110 activities were affected by data loss problems and we ended up discovering 298 activities affected by data loss problems. Interestingly, the probabilistic data-loss-revealing action contributed revealing data loss in 158 activities already reported by the systematic data-loss-revealing action.

We manually investigated the 50 cases of known data loss faults that have not been revealed by DLD (27 cases in the benchmark

and 23 cases retrieved online). We isolated three main reasons why data loss problems have been missed.

- *Low probability sequences*: revealing these data loss failures requires the generation of an event sequence that has a low probability to be generated, due to the length of the sequence and/or the large number of actions that can be generated at every step of the testing process.
- *Environment setup*: the detection of these data loss problems requires a specific set up of the environment. For instance, a data loss affecting the DOCUMENT VIEWER app requires the presence of a document to be revealed. These faults could be potentially revealed with additional effort in the setup of the app under test.



**Figure 5: Percentage of spurious oracle violations returned per app.**

- *Unsupported actions*: these data loss failures are impossible to reveal with DLD because they require the execution of operations that are outside the scope of DLD. For instance, detecting one of the data loss faults in the TASKS ASTRID To-Do List CLONE app requires to temporarily exit from the app and take a picture with the device camera, which cannot be done with DLD.

Overall, we found 41 data loss problems that simply have low probability to be revealed, 4 data loss problems that require a proper environment setup, and 5 data loss problems that require a more extensive exploration ability to be revealed. Interestingly, several faults might be potentially revealed by just executing DLD for a longer time, or by refining the DLD exploration strategy, so that specific combinations of actions are generated. However, generating complex and long combinations of actions can be challenging.

Finally, only 38 out of 298 activities affected by data loss faults produced crashes, which confirms the need of specific oracles to deal with these problems.

**4.5.2 RQ1.2 - What is the rate of the spurious oracle violations reported by DLD?** DLD performed well in terms of spurious oracle violations: it reported only 1 activity with spurious data loss only every 4 tested apps, which indicates that DLD is precise and seldom annoys testers with false alarms.

Figure 5 shows the percentage of spurious oracle violations returned per app. The percentage ranges between a min of 0% and a max of 24%, with a mean value of 10.4% and a median value of 2.7%. Indeed, DLD produces a limited percentage of spurious violations (less than 11% for 75% of the apps) that can be feasibly inspected by engineers when analyzing the output produced by the technique. The 5 outliers reported in Figure 5 correspond to apps with elements difficult to handle, such as timers and progress bars, that can be the source of an abnormal number of spurious violations due to spontaneous changes happening concurrently with the double rotations. We discuss the source of these spurious violations in RQ3.

## 4.6 RQ2 - Is DLD more effective than state-of-the-art techniques?

This research question compares DLD to both ALARic [24] and Quantum [33]. ALARic represents the case of an *alternative automated approach* to reveal data loss faults, while Quantum represents

the case of an approach that can benefit from a *manually generated model* to generate data loss-revealing test cases.

**4.6.1 RQ2.1 - What is the relative effectiveness of DLD, ALARic, and Quantum?** The comparison to ALARic studies the effectiveness of the test generation strategy defined in DLD, as described in Section 3, to ALARic, which uses a random (non-biased) exploration and a concrete states representation.

We executed ALARic three times for 3 hours each time, as done for DLD, and reported the results in Table 2 (column ALARic). Note that we excluded from the comparison the apps that have been tested with DLD exploiting an ad-hoc setup, since it would lead to an unfair comparison to ALARic, which does not implement this feature. Table 2 shows with grey background the cases where a technique outperforms the other.

DLD significantly outperformed ALARic in terms of data loss discovery capability. In fact, ALARic revealed 71 activities affected by data loss faults, while DLD revealed 189 faulty activities, releasing a 2.7X factor of improvement. ALARic found 19 data loss faults of the benchmark, 8 online data loss faults, and 50 new data loss faults. While DLD revealed 73 of the data loss faults in the benchmark (3.8X improvement factor), 35 online data loss faults (4.4X improvement factor), and 132 new data loss (2.6X improvement factor). Overall, DLD revealed *significantly more* data loss faults than ALARic.

DLD performed better than ALARic also in terms of spurious data loss. In fact, DLD produced spurious data loss only for 11 activities, while ALARic produced spurious data loss for 31 of the activities.

In summary, DLD has been significantly more effective than ALARic with the studied apps.

Since Quantum is not publicly available, we could not compare Quantum to DLD on our set of apps. We thus executed DLD for 3 hours on the same apps used in the evaluation of Quantum [33] and compared the results. We limited the experiment to 4 of the 6 apps used to evaluate Quantum since for 2 apps it was impossible to retrieve the same version used in the original study.

Since we do not know the manual effort that was necessary to manually define the models used by Quantum, it is hard to setup a fair comparison among the two approaches. However, the obtained results can still offer useful insights about the relative effectiveness of the two approaches.

**Table 3: Comparison between Quantum and DLD.**

App (Version)	Quantum (with manual model)		DLD	
	data loss	spurious violation	data loss	spurious violation
OpenSudoku (1.1.5)	3	2	5	1
Nexes Manager (2.1.8)	7	2	11	1
VuDroid (1.4)	2	0	2	0
K9Mail (4.317)	4	1	15	2

Table 3 shows the distinct data loss and spurious violations reported by Quantum and DLD. Although DLD cannot benefit from a manual model, its activity has been more effective in revealing a number of data loss faults compared to Quantum. Of course, we

do not know if the data loss faults revealed by DLD include all the data loss faults revealed by Quantum. It might be the case that DLD cannot reach some areas of the app under test that can be reached with the manual model. However, the results suggest that DLD is quite effective even compared to techniques exploiting manual models.

**4.6.2 RQ2.2-What are the main factors that determine the effectiveness of DLD?** To investigate the reason of the difference in the performance between DLD and ALARic, we studied the reason why ALARic failed to reveal faults revealed by DLD. In particular, we counted the number of faulty activities not reached by ALARic and the number of faulty activities reached by ALARic without revealing the data loss fault. This produced the following results:

- ALARic does not reach 52% of the faulty activities revealed by DLD only, that is, approximatively half of the additional data loss faults revealed by DLD are due to a better exploration strategy,
- ALARic reaches the faulty activity without revealing the data loss for 48% of the faulty activities revealed by DLD only. In a nutshell, the systematic fine-grained testing of the states implemented in DLD is more effective than Alaric's strategy,
- 12 of the faulty activities missed by Alaric require a snapshot-based oracle to be revealed, but only 4 of them are reached by ALARic.

#### 4.7 RQ3 - What is the tradeoff between the snapshot- and property-based oracles?

This research question investigates the complementarity between the snapshot-based and the property-based oracles. We already discussed in Section 3.3 the qualitative differences between these two types of oracles, and reported examples of data loss faults that could be detected by one type of oracle only. Here, we assess quantitatively the impact of each class of oracles, on both the revealed data loss faults and the spurious oracle violations.

Figure 6 shows the percentage of data loss faults detected and the number of spurious oracle violations produced by one-strategy only, either snapshot-based or property-based, or both of them. In the case of the spurious violations we have an additional category that is the violations caused by slow activity recreation after screen rotation, as anticipated in the Section discussing RQ1.

None of the two approaches have been able to reveal every data loss problem. A large proportion of the failures (73.1%) have been detected by both oracles, which implies that most of the data loss faults cause both properties that lose their values and visible issues on the app. However, there are yet 26.9% of the faults that require a specific type of oracle to be revealed.

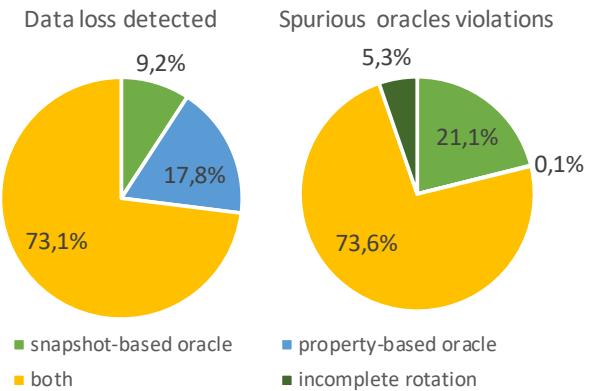
In terms of absolute failure discovery ability, both oracles have been effective, with the property-based and snapshot-based oracles revealing 90.9% and 82.3% of the failures, respectively.

A small number of the spurious oracle violations (5.3%) is caused by a slow activity recreation, which causes the oracles to retrieve incorrect state information. This percentage can be reduced or eliminated by carefully tuning the timing of the oracles.

Interestingly, the property-based oracle is also more effective in terms of spurious violations reported. In fact, only 0.1% of the spurious violations are produced uniquely by the property-based

oracle, while 21.1% of the spurious violations are produced uniquely by the snapshot-based oracle. The largest proportion of the spurious violations (73.6%) are produced by both the strategies.

Although the snapshot-based oracle produces more spurious violations than the property-based oracle, these violations seldom cause correct activities to be reported to the tester (1 activity every 4 apps), thus it is relatively detrimental to use it in the testing process. On the contrary, including it in the analysis increases the number of revealed data loss faults by 9.2%, which is a non-trivial increase of the failure discovery ability of DLD.



**Figure 6: Percentage of detected data loss failures and spurious violations per oracle strategy.**

#### 4.8 RQ4 - Are data loss faults relevant to developers?

Finally, we investigated if data loss faults are relevant to app developers. To this end, for each app in the benchmark, we identified and downloaded the latest version of the same app. We executed again DLD for 9 hours (three 3-hours runs) on each app and revealed 195 data loss faults that still affect these apps nowadays. We finally submitted a bug report online for each revealed data loss.

Up to the time of the conference deadline, we received feedback for 98 of the reports submitted online. Developers confirmed the bugs for 88 reports (90% of the reports with a feedback). Only in 10 cases developers rejected the report advocating that the fault was a framework fault, claiming the fault was not reproducible, or giving no explanation. We can thus conclude that the revealed data loss faults are significant to the developers.

In 33 of 88 cases developers claimed that the cost of fixing these bugs might be too high compared to their impact on the app. This decision of course depends on the specific consequence of the data loss and the complexity of the activity that is affected by the fault. This also suggests that the definition of an automatic repair strategy [15] that can address data loss faults could be extremely beneficial to improve the cost-effectiveness of the bug fixing process.

#### 4.9 Threats to Validity

The main internal threats to validity about our study is the manual work done to identify the spurious violations among the data loss faults reported in the evaluation. Distinguishing a genuine data

loss from a spurious one is however quite simple, as also confirmed by the bug reports submitted online to developers that have been almost all accepted, with rejections due to faults that were considered outside the boundary of the tested app or behaviors that could be considered acceptable although not ideal.

The main external threats to validity concerns with the generalization of the results. The significant number of faults and apps considered mitigates this threat. The fact that we repeated the evaluation with the most recent versions of the apps revealing again many data loss faults is a further mitigation factor.

Concerning the comparison between DLD and ALARic, the consistency of the results across every app that has been tested is a strong factor in favour of the generality of the results. The results of the comparison between DLD and Quantum cannot be generalized due to the limited size and the setup of the experiment, however they still provide useful insights about the effectiveness of DLD.

## 5 RELATED WORK

Several techniques covering a wide range of approaches are available to generate test cases for Android apps. For instance, Monkey generates test inputs fully randomly without interpreting the GUI of the app under test [12]. A<sup>3</sup>E systematically generates test inputs following a depth-first strategy [3]. DroidBot [19] and Stoat [31] build a state-based model of the system under test and generate test cases exploiting information about the events already tested in the visited states. Sapienz uses evolutionary algorithms to generate test cases [20]. While these approaches revealed several interesting faults in both open source [6] and industrial applications [32], they are ineffective against data loss problems (and also against most non-crashing failures [33]). In fact, they neither include operations that cause stop-start events nor they are equipped with oracles that can detect non-crashing data loss failures, which account for the majority of the failures as reported in our evaluation.

Thor [1] can augment existing test suites with neutral sequences of operations to reveal additional failures. The injected sequences concern with the audio service, the connectivity, and the lifecycle of the activities, which may reveal data loss faults. Similarly, when a user-generated model of the app under test is available, Quantum [33] can generate tests that may reveal data loss, as reported in a small-scale evaluation. Differently from these approaches, DLD directly generates the test cases and requires neither an initial test suite nor a model of the app under test, retaining a high effectiveness as demonstrated in the comparison to ALARic and Quantum.

CrashScope [22] and AppDoctor [17] can generate tests that may reveal data loss problems, but their effectiveness is limited to crashing faults, which represent the minority of the cases.

ALARic [2] is a mostly random test case generation technique that similarly to DLD exploits double screen rotations to reveal data loss faults. However, the biased exploration strategy, the enabledness state abstraction, and the ad-hoc data-loss-revealing actions used by DLD outperformed ALARic in our evaluation.

When the source code of the app is available, a static analysis technique such as KREFinder [28] can be used to reveal data loss problems. However, as most static analysis techniques, KREFinder suffers scalability issues and is likely to report many spurious violations. On the contrary, the effectiveness of DLD does not depend

on the complexity and size of the source code and seldom reports spurious data loss.

The oracle strategies presented in this paper relate to the work on metamorphic testing [27]. Metamorphic testing exploits metamorphic relations, which are relations on multiple executions of the software, to check the correctness of the observed behavior. The neutral sequences of operations that DLD uses to reveal data loss problems can be seen as a specific class of metamorphic relations that relate executions with and without these sequences.

Relations between executions, like the ones used in this paper to reveal data loss problems, have been also used to heal executions, for instance to produce automatic workarounds [5]. Although neutral sequences of events can be potentially used to obtain workarounds, the ones used in this paper can be hardly used to heal executions since they are often the cause of faults, as reported in the evaluation.

Finally, faults in Android apps, including data loss faults, could be addressed with healing techniques. However, not many healing approaches can work in the Android environment. Azim *et al.* defined a technique that can disable functionalities that are not working properly [4]. While this approach might be exploited to prevent data loss failures, it also reduces the set of functionalities available to users. DataLossHealer is a healing solution designed to mitigate the impact of data loss faults in the field [25]. Although it might prevent some data loss faults, it has various drawbacks. For instance, it introduces overhead to save and restore data in presence of data loss faults, it can address only some data loss, and it requires rooting the device. DLD delivers a more effective solution revealing data loss faults upfront before the app is released.

## 6 CONCLUSIONS

Android apps must be designed to deal with stop-start events, which are external events that may interrupt the execution of the running activity. When one of these events is generated, the foreground Android activity might be destroyed and later recreated. To avoid losing useful data during this process, apps must explicitly implement the logic necessary to save the data, when the activity is destroyed, and restore the saved data, when the activity is recreated. Unfortunately, this logic is often faulty [1, 2, 24, 26].

This paper presents Data Loss Detector (DLD), an automatic test case generation technique designed to reveal data loss faults. DLD exploits an exploration strategy biased towards the discovery of new app states, data-loss-revealing actions, and two dedicated oracle-based strategies to automatically reveal data loss problems.

In our evaluation with 110 data loss faults affecting 54 app releases, DLD outperformed ALARic [24] and performed well in comparison to Quantum when instructed with a manual model of the app under test. Overall, DLD revealed 298 activities affected by data loss faults, which is a clear indicator of the effectiveness of the approach and pervasiveness of the problem.

We are now working on the definition of automatic program repair solutions for data loss faults.

## ACKNOWLEDGMENTS

We would like to thank Vincenzo Riccio, Domenico Amalfitano and Anna Rita Fasolino for sharing their implementation of ALARic with us.

## REFERENCES

- [1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.
- [2] Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Software Testing, Verification and Reliability* 28, 1 (2018), e1654.
- [3] Md. Tanvirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*.
- [4] Md. Tanvirul Azim, Iulian Neamtiu, and Lisa M. Marvel. 2014. Towards Self-healing Smartphone Software via Automated Patching. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
- [5] Antonia Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2015. Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications. *ACM Transactions on Software Engineering and Methodologies (TOSEM)* 24, 3 (2015).
- [6] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [7] Guido De Caso, Victor Braberman, Diego Garberovetsky, and Sebastián Uchitel. 2011. Program Abstractions for Behaviour Validation. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
- [8] Guido De Caso, Victor Braberman, Diego Garberovetsky, and Sebastian Uchitel. 2013. Enabledness-based Program Abstractions for Behavior Validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013), 25:1–25:46.
- [9] Android Developers. [n.d.]. *Fragments*. <https://developer.android.com/guide/components/fragments>
- [10] Android Developers. [n.d.]. *Saving UI States*. <https://developer.android.com/topic/libraries/architecture/saving-states.html>
- [11] Android Developers. [n.d.]. *Services overview*. <https://developer.android.com/guide/components/services>
- [12] Android Developers. [n.d.]. *UI/Application Exerciser Monkey*. <https://developer.android.com/studio/test/monkey>
- [13] Android Developers. [n.d.]. *Understand the Activity Lifecycle*. <https://developer.android.com/guide/components/activities/activity-lifecycle>
- [14] JS Foundation. [n.d.]. *Appium*. <http://appium.io/>
- [15] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering (TSE)* 45, 1 (2019), 34–67.
- [16] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST)*.
- [17] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*.
- [18] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2019. Characterizing Android-specific Crash Bugs. In *Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*.
- [19] Yuanchun Li, Yang Ziyue, Guo Yao, and Chen Xiangqun. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *Proceedings of the International Conference on Software Engineering Companion (ICSE)*.
- [20] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.
- [21] Mobilloud. 2019. People Spent 90% of Their Mobile Time Using Apps in 2019. <https://www.mobilloud.com/blog/mobile-apps-vs-the-mobile-web/>. [Online; accessed January 2020].
- [22] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.
- [23] RescueTime:blog. 2019. Screen time stats 2019: Here's how much you use your phone during the workday. <https://blog.rescuetime.com/screen-time-stats-2018/>. [Online; accessed January 2020].
- [24] Vincenzo Riccio, Domenico Amalfitano, and Anna Rita Fasolino. 2018. Is this the lifecycle we really want?: an automated black-box testing approach for Android activities. In *Proceedings of the International Workshop on User Interface Test Automation, and Workshop on TESting Techniques for event BasED Software (ISSTA/ECOOPWorkshops)*.
- [25] Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. 2016. Healing Data Loss Problems in Android Apps. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*.
- [26] Oliviero Riganelli, Marco Mobillo, Daniela Micucci, and Leonardo Mariani. 2019. A Benchmark of Data Loss Bugs for Android Apps. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*.
- [27] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering (TSE)* 42, 9 (2016), 805–824.
- [28] Z. Shan, T. Azim, and I. Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.
- [29] Statcounter. 2020. Mobile Operating System Market Share Worldwide - December 2019. <http://gs.statcounter.com/os-market-share/mobile/worldwide>. [Online; accessed January 2020].
- [30] Statista. 2020. Number of available applications in the Google Play Store from December 2009 to December 2019. <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>
- [31] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*.
- [32] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
- [33] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*.

# Reinforcement Learning Based Curiosity-Driven Testing of Android Applications

Minxue Pan\*

mxp@nju.edu.cn

State Key Lab for Novel Software  
Technology, Nanjing University  
Software Institute, Nanjing University  
Nanjing, China

An Huang

ha@smail.nju.edu.cn

State Key Lab for Novel Software  
Technology, Nanjing University  
Department of Computer Science and  
Technology, Nanjing University  
Nanjing, China

Guoxin Wang

njuwgx@smail.nju.edu.cn

State Key Lab for Novel Software  
Technology, Nanjing University  
Department of Computer Science and  
Technology, Nanjing University  
Nanjing, China

Tian Zhang\*

ztluck@nju.edu.cn

State Key Lab for Novel Software  
Technology, Nanjing University  
Department of Computer Science and  
Technology, Nanjing University  
Nanjing, China

Xuandong Li

lxd@nju.edu.cn

State Key Lab for Novel Software  
Technology, Nanjing University  
Department of Computer Science and  
Technology, Nanjing University  
Nanjing, China

## ABSTRACT

Mobile applications play an important role in our daily life, while it still remains a challenge to guarantee their correctness. Model-based and systematic approaches have been applied to Android GUI testing. However, they do not show significant advantages over random approaches because of limitations such as imprecise models and poor scalability. In this paper, we propose Q-testing, a reinforcement learning based approach which benefits from both random and model-based approaches to automated testing of Android applications. Q-testing explores the Android apps with a curiosity-driven strategy that utilizes a memory set to record part of previously visited states and guides the testing towards unfamiliar functionalities. A state comparison module, which is a neural network trained by plenty of collected samples, is novelly employed to divide different states at the granularity of functional scenarios. It can determine the reinforcement learning reward in Q-testing and help the curiosity-driven strategy explore different functionalities efficiently. We conduct experiments on 50 open-source applications where Q-testing outperforms the state-of-the-art and state-of-practice Android GUI testing tools in terms of code coverage and fault detection. So far, 22 of our reported faults have been confirmed, among which 7 have been fixed.

\*Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397354>

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Android app testing, reinforcement learning, functional scenario division

### ACM Reference Format:

Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397354>

## 1 INTRODUCTION

The proliferation of mobile devices and apps makes a huge impact on our daily life. Statistics [50] show that an average user spends more than 2 hours a day on mobile apps. However, it remains a challenge to guarantee apps' quality because of the large combinatorial space of possible events and transitions. A daily used app usually contains plenty of interfaces and executable events, which makes it time-consuming to explore all its states, let alone particular functionalities that can be accessed only in specific conditions. Different strategies have been applied to automated testing of Android applications, which, unfortunately, still need improvement [10].

Random strategies [21, 31] generate pseudo-random events to fuzz the application under test. Monkey [21], regarded as the state-of-practice testing tool, is a typical example of this strategy. Despite the wide adoption in practical development, its shortcomings are quite obvious. It is common for Monkey to generate futile events like clicking a non-interactive area of the screen that makes no changes to the current state. Also, the testing is unbalanced and some hard-to-reach functionalities may never be explored.

Model-based strategies [2, 5, 46] generate test cases according to application models which are constructed with a static or dynamic

approach. In this case, high-quality models are extremely important in order to achieve a good testing result. However, as illustrated above, it is challenging to explore all the states of an Android application. What's more, it is almost impossible to precisely model the apps' behavior. For example, a welcome interface is common in nowadays applications which will appear only when the app is opened the first time. This information often cannot be captured, making the generated event sequences always contain events in the welcome interface, which is inconsistent with apps' actual behavior and is highly detrimental to the testing effectiveness.

Systematic strategies [4, 27, 32] use sophisticated techniques such as symbolic execution to provide specific inputs for targeted application behavior. These strategies are mainly designed to reveal typical functionalities that are hard to execute with other strategies, but they are less scalable and often perform worse in overall testing metrics like code coverage and bug revelation.

Machine learning techniques are starting to find application in Android GUI testing, too. Recently, a few works [1, 26, 48] utilize reinforcement learning, specifically Q-learning which can benefit from both random and model-based approaches, to guide the testing progress. The Q-table, which records each event's value, along with the propagation property of Q values can partly take the place of models to store testing related information in a light way. It also helps to avoid the inconsistency problem between models and apps' actual behavior which is an advantage of random testing. However, existing works do not fully unleash the ability of reinforcement learning in Android testing. Take the reward giving process, which is key to reinforcement learning, as an example. Prior works tend to calculate the differences between two states (the ones before and after an event is executed) to determine the reward. If two states are quite different, the testing tools will continuously give a large reward which results in frequently jumping between them even if they have been over explored. Although some of the reward calculating functions take the executing frequency into consideration, the problem arises again when most of the events have been executed several times.

To tackle the aforementioned challenges and to unleash machine learning's potential in Android GUI testing, we propose a novel approach, Q-testing, based on reinforcement learning. The strategy of Q-testing is called curiosity-driven exploration which guides testing towards states that it is curious about. More precisely, Q-testing maintains a set, which acts as memory, to record part of the previously visited states. The reinforcement learning reward is calculated according to the differences between the current state and those recorded in memory. Different from prior works, curiosity-driven exploration is a dynamically adaptive strategy. By adaptive, we mean that it can spot changes in the importance of states and will continuously adjust the reward for a certain event.

In order to improve test efficiency, we novelly propose a neural network to divide different states at the granularity of functional scenarios. Q-testing uses this module to compare states and calculate rewards. With its help, Q-testing will preferentially make effort to covering different functionalities which helps to rationally allocate limited testing time and will result in the rapid growth of code coverage in a short time. We collect more than 6k samples to train the model and it can be used not only in reinforcement learning.

Other tasks including state compression and code recommendation may also benefit from it.

Q-testing also involves testing strategies that are specifically designed for Android applications. We resolve RecyclerView [19] and ListView [18] in a manner to forbid the waste of time in unnecessary testing. We also take system-level events into consideration and it helps Q-testing find some complicated bugs.

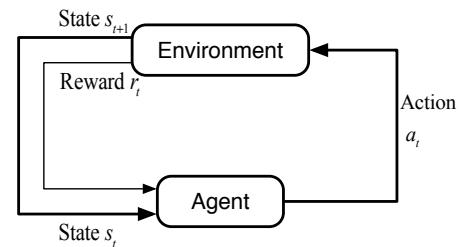
In this paper, we make the following main contributions:

- We propose a novel curiosity-driven exploration strategy based on reinforcement learning to guide Android automated testing.
- We collect samples and train a neural network, which can effectively divide different states at the functional level in an efficient manner.
- We implement a tool and conduct a large-scale experiment. Results show that our approach outperforms existing ones in terms of both code coverage and fault detection. The tool and experimental data are publicly available<sup>1</sup> to facilitate future researches.

The remainder of the paper is structured as follows. Section 2 gives an introduction to Q-learning. Section 3 describes our reinforcement learning based testing strategy. Section 4 presents the state comparison module. Section 5 presents the experiment results. Section 6 surveys related work and Section 7 makes a conclusion.

## 2 Q-LEARNING

Q-learning [49] is a form of model-free reinforcement learning whose goal is learning how to map situations to actions so as to maximize a numerical reward signal in an unknown environment. The two most distinguishing features of reinforcement learning are trial-and-error search and delayed reward. The agent must try different actions to discover which may yield the most reward, and actions may affect not only the immediate reward but also subsequent states along with future rewards.



**Figure 1: Markov Decision Process**

The problem of reinforcement learning can be formalized with ideas from dynamic systems theory, specifically the Markov decision process, or MDP. MDP can be defined as a 4-tuple  $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$  where  $\mathcal{S}$  represents the set of states and  $\mathcal{A}$  represents the set of actions. As depicted in Figure 1, the agent and the outside environment interact at each of the discrete time steps of a sequence. At each time step  $t$ , the agent selects and executes an action,  $a_t \in \mathcal{A}$ ,

<sup>1</sup><https://github.com/anlalalu/Q-testing>

based on its observation of the state,  $s_t \in \mathcal{S}$ . After that, the agent will move to a new state,  $s_{t+1} \in \mathcal{S}$ , and receive an immediate reward,  $r_t \in \mathcal{R}$ , at the same time. The variables  $r_t$  and  $s_t$  have well-defined probability distributions dependent only on preceding state and action. That is,  $s_{t+1} \sim \mathcal{P}(s_t, a_t)$  and  $r_t \sim \mathcal{R}(s_t, a_t)$ . The return which represents the cumulative reward is often defined as  $R_t = \sum_{t>0} \gamma^{t-1} r_t$ , where future rewards are discounted by a factor of  $\gamma \in [0, 1]$ .

Reinforcement learning usually involves Q value functions to estimate how good it is to perform an action in a certain state. This action-value function returns the expected cumulative reward of a sequence of actions which starts from action  $A_t$  in state  $S_t$  and thereafter following policy  $\pi$ :

$$Q^\pi(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t] \quad (1)$$

With the Bellman equation, we can express the relationship between the value of a state-action pair and its successor state-action pair:

$$Q^\pi(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, \pi_{s_{t+1}})] \quad (2)$$

Q-learning uses equation 2 to estimate each state-action pair. If the states and actions are discrete and finite, the pairs can be represented in a tabular form where all pairs will be given an initial value. Every time an action is executed, the relating state-action value will be updated:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q^*(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (3)$$

$Q^*(s_{t+1}, a_{t+1})$  represents the maximum cumulative reward that can be achieved from state  $s_{t+1}$ . The  $\alpha$  is the learning rate which is between 0 and 1. As we can see in this equation, the value of subsequent state-action pair will be propagated to influence preceding pairs. There is a strict proof that the estimator will converge to the true value if the environment is explored sufficiently. Whenever Q-learning estimates the values precisely, we can easily find the optimal policy simply by executing the action which has the highest value at every state.

Q-learning provides agents with the capability of learning to act optimally in Markovian environments without requiring them to build models of the environments. We observe that an Android application testing process can be viewed as a Markov Decision Process: by giving rewards when test actions lead to new states of applications, the entire testing should be able to learn to cover more functionalities of the applications. This inspires us to apply Q-learning to Android automated testing.

### 3 Q-LEARNING BASED ANDROID TESTING

The Android testing task can be viewed as a Markov Decision Process which makes it possible for reinforcement learning to play a role. We design exploration strategies based on Q-learning to guide the testing tool towards unrevealed and unfamiliar functionalities.

#### 3.1 Approach Overview

The workflow of Q-testing is depicted in Figure 2. Analogous to the process of MDP, Q-testing interacts with the outer environment, the application under test (AUT), during testing. In each cycle, Q-testing

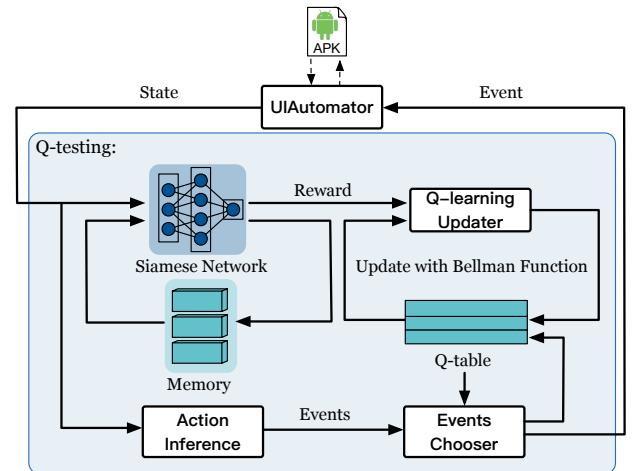


Figure 2: Q-testing Workflow

first observes the application's current state  $s_t$  with UIAutomator [20]. Then the state  $s_t$  is compared with part of prior observed states. The states are stored in a buffer which acts like Q-testing's memory. A neural network is trained to extract states' features and conduct the comparison. If state  $s_t$  is similar to any of the states in memory, the comparator will give a small reward. Otherwise, the module will give a large reward and state  $s_t$  will be added to the memory buffer. The reward is used to update the value of the state-action pair  $(s_{t-1}, a_{t-1})$ . All the state-action pairs' values are stored in Q-table and their values are updated with equation 3. Every time a new state is reached, Q-testing will add related state-action pairs into Q-table and initialize it with a large value to encourage the execution of new events. After the updating of Q value, Q-testing will infer the executable events from the GUI hierarchy of  $s_t$  and choose an event  $a_t$  with reference to Q-table. In most instances, the event with the highest value will be chosen. After the execution, the AUT will respond to  $a_t$  and another cycle starts.

#### 3.2 Formulating Android Testing as MDP

In our approach, an Android GUI testing problem is mathematically formalized as an MDP that can be defined with a 4-tuple,  $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ . How we define  $\mathcal{S}$ ,  $\mathcal{A}$ ,  $\mathcal{P}$  and  $\mathcal{R}$  for the Android GUI testing task will make an impact on the testing effectiveness and efficiency. In this paper, we adopt the following formulation.

**S: States.** Prior Android automated testing works [1–3, 5, 6, 23, 46, 48, 53] adopt different criteria to abstract applications' states. Baek et al. [6] conducts experiments to identify the effect of comparison criteria on Android GUI testing and the result indicates that finer comparison granularity will benefit testing by achieving higher code coverage and finding more bugs. For this reason, we adopt the widget composition as a comparison criterion. Specifically, Q-testing uses UIAutomator to extract the GUI hierarchy. UIAutomator dumps the information of the widgets that are contained in the current interface. The widgets are arranged in a tree structure where non-leaf nodes represent layout widgets and leaf nodes represent executable widgets. Q-testing ignores some of the widgets' attributes including text to avoid state explosion. This

detailed information will be disruptive to the update of Q-table and will reduce testing efficiency. In brief, our state  $s_t$  is defined as a combined state  $(w_1, w_2, \dots, w_n)$  where  $w_i$  is the state of widget contained in  $s_t$  and it is defined by selected attributes which include an index attribute to describe the widget's position in the GUI hierarchy tree.

**A: Actions.** We formulate user interaction events in apps as actions in MDP, and we do not distinguish events and actions in this paper. Similar to previous work [6, 23, 46], Q-testing infers executable events in the current state by analyzing the dumped widget hierarchy and corresponding attributes (e.g., clickable, scrollable).

As each event is associated with a specific state, this enables us to also use the state-action pairs to represent an event executable in a state of the application. Equation 4 describes the  $\epsilon$ -greedy policy that is utilized by Q-testing to select the next event. Q-testing selects the event with the highest Q value with probability  $1 - \epsilon$  and selects a random event with probability  $\epsilon$ . In order to trigger intricate bugs, Q-testing also takes system-level events into consideration. The value of  $\epsilon$  can be adjusted and its default setting is 0.2 in Q-testing.

$$getAction(s) = \begin{cases} \arg \max_a Q(s, a), & 1 - \epsilon \\ \text{random UI event}, & \frac{1}{2}\epsilon \\ \text{random system event}, & \frac{1}{2}\epsilon \end{cases} \quad (4)$$

**P: Transition Function.** The transition function depicts which state the application will transit to after an event is executed. It is determined by the AUT and we cannot make any change to it. What should be emphasized is the non-deterministic transition of which the result of executing an event can be changed by the states of internal variables. Prior model-based approaches usually ignore this kind of transition due to the limited ability of their modeling processes. Q-testing updates Q values with the Bellman function of which the value of a state-action pair  $(s, a)$  is influenced by all the states that can be reached from  $s$ . Therefore, all the transition information is considered when the values are updated.

**R: Reward.** Q-testing receives a reward every time it executes an event. We propose a policy to determine the reward, which contributes to our exploration strategy. The details are illustrated in the following subsections.

### 3.3 Exploration Strategies

Q-testing employs the curiosity-driven strategy to test the AUT. Curiosity, which encourages the agent to explore the outer environment, is studied in reinforcement learning tasks [9, 25, 38, 43] to solve the problem of sparse reward. The curiosity-driven strategy proposed by Q-testing can guide the testing tool to explore unfamiliar states in order to cover codes and find bugs with high efficiency.

Algorithm 1 describes the algorithm of the curiosity-driven exploration strategy. Specifically, Q-testing maintains a state buffer to store part of previously visited states (line 2). This buffer acts like a memory and it helps Q-testing to find out the states that it is unfamiliar with, in other words, curious about. Q-table stores all the state-action pairs grouped by activities (denoted as  $act$ ) along with their values. Every time a state is reached, Q-testing will directly look up the Q-table to find out the executable actions. If the state is reached for the first time, Q-testing will infer executable actions

---

**Algorithm 1** Curiosity Driven Testing

---

```

Input: the app under test  $AUT$ , execution time  $t$ , learning rate  $\alpha$ , similarity threshold  $threshold$ 
1:  $Q \leftarrow \emptyset$                                  $\triangleright$  initialize Q-table
2:  $M \leftarrow \emptyset$                                  $\triangleright$  initialize memory buffer
3:  $s_{t+1}, act_{t+1} \leftarrow \text{getCurrentState}(AUT)$ 
4:  $v_{t+1} \leftarrow \text{extractVector}(s_{t+1})$ 
5:  $M \leftarrow M \cup (act_{t+1}, v_{t+1})$ 
6: while  $\neg \text{timeout}(t)$  do
7:    $s_t \leftarrow s_{t+1}$ 
8:    $A \leftarrow \text{getOrInferEvents}(Q, s_t)$ 
9:    $a_t \leftarrow \text{getAction}(Q, s_t)$ 
10:   $s_{t+1}, act_{t+1} \leftarrow \text{execute}(AUT, a_t)$ 
11:   $v_{t+1} \leftarrow \text{extractVector}(s_{t+1})$ 
12:  for all  $(act_{t+1}, v) \in M$  do
13:     $d \leftarrow \text{calculateDistance}(v, v_{t+1})$ 
14:     $similarity \leftarrow \min(similarity, d)$ 
15:  end for
16:  if  $similarity \geq threshold$  then
17:     $r_t \leftarrow smallReward$ 
18:  else
19:     $M \leftarrow M \cup (act_{t+1}, v_{t+1})$ 
20:     $r_t \leftarrow largeReward$ 
21:  end if
22:   $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q^*(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$ 
23: end while

```

---

from the GUI hierarchy and initializes them in Q-table (line 8). The initial value for all the unexecuted actions is set to 1000, which is a large number, to encourage executing of new actions and it will be assigned to 0 after executed. Q-table acts like a lightweight model that speeds up the testing process, and what is more, it avoids the problems caused by imprecisely modeling. Q-table also guides the selection of next executed action as depicted in equation 4 (line 9). After an action  $a_t$  is executed from state  $s_t$ , the AUT will respond to it and transit to a state  $s_{t+1}$  (line 10). Then  $s_{t+1}$  is compared with those stored in memory and a reward will be given according to their differences (lines 11-21). The curiosity-driven strategy will steer testing towards unfamiliar states by giving a large reward when reaching a state that is, to a certain extent, different from all the remembered states. Q-testing will also update its memory by adding the unfamiliar states into the buffer and the next time the state is visited, it will not be attractive anymore. We set the reward to be 500 for actions leading to unfamiliar functionalities, which will improve the action's value according to the Bellman equation. As for actions that make little contribution to Q-testing exploring new parts of the AUT, the reward is set as -500. The reward values are configurable, however, these default values work well on different types of applications, as shown in the experiments.

It is worth emphasizing that the strategy benefits from the propagation property of Q value (line 22), which makes it possible to guide Q-testing towards valuable states no matter what values of the current state is. Suppose that Q-testing executes an event  $a_t$  and the AUT transits to state  $s_{t+1}$  that is similar to one of the states in memory set. Then a -500 reward will be given which may decrease the value of  $a_t$ . However, if  $s_{t+1}$  is a valuable state that can lead

to unexplored scenarios, then  $Q^*(s_{t+1}, a_{t+1})$ , which is the maximum of the Q-values of all state-action pairs in state  $s_{t+1}$  stored in the Q-table, will be large. This results in that the value of  $a_t$  will still be improved, since  $\gamma$  is set to 0.99 in our implementation to strengthen the impacts of new states. This makes the unexplored functionalities more likely to be explored.

Additionally, since the current state has to be compared with all those in memory, several measures are taken to improve the comparing efficiency. First, Q-testing stores feature vectors rather than the GUI hierarchy information in memory (line 11, 19). Similarity can be computed between two vectors with Manhattan distance function without extracting features repeatedly. Second, the vectors are grouped by activities and a vector only has to be compared with those in the same activity (line 12). Third, We propose a coarse-grained criterion to divide states and it benefits the testing process in several ways. On the one hand, it restricts the number of states stored and further reduces time spent in comparing states. On the other hand, it encourages Q-testing to explore states with much more differences and this will result in high code coverage in a short time since the states with a huge difference are often bound to different codes. Other details of this comparison criterion will be described in section 4.

### 3.4 Android-Specific Strategies

Despite the curiosity-based strategy, Q-testing also takes advantage of other strategies specifically designed based on characteristics of Android applications, including the special treatment of special widgets and the injection of system events.

RecyclerView [19] and ListView [18] are two Android widgets that are able to display a collection of views in a scrollable manner. In most cases, these widgets will contain a lot of items and users can continuously view new items by scrolling. The clicking of these items often leads to similar screens (with the same GUI hierarchy and different contents) and can only trigger the same codes. Existing works make no consideration on these widgets and may waste plenty of time testing different items. Q-testing analyzes the result of relative events. If several items in a RecyclerView lead to similar states, Q-testing will ignore the items except the first one. This strategy can save a lot of time since RecyclerView and ListView are quite common in our daily used applications.

System events can benefit testing by triggering intricate bugs. Similar to Stoat [46], Q-testing takes three kinds of system-level events, including user actions (e.g., screen rotation, phone calls), broadcast messages [17] (e.g., switch into or out of airplane mode) and application-specific events which can be extracted from Android manifest files, into consideration. Q-testing executes system-level events randomly during its exploration. These events are not included in Q-table since the number of possible system events is very large. Once they are added, Q-testing may try to execute all of them in every state.

## 4 SCENARIO DIVISION MODULE

We propose a new state comparison criterion at coarse granularity and to determine the reward in the curiosity-driven exploration strategy. The module is able to determine whether two states are in

the same functional scenario. Reward determined by the comparison module will guide Q-testing to firstly cover different functional scenarios. The coarse granularity states division helps to allocate limited testing time rationally by preventing Q-testing from falling into the several same scenarios at the beginning. This will also result in high code coverage in a short period of time since different scenarios are usually bound to different codes. More specifically, a great number of codes (e.g., the codes used to initialize the layout and interaction events) may be covered the first time a functional scenario is reached, which is determined by the GUI driven feature of Android framework.

### 4.1 The Task of Scenario Division

The definition of a scenario for our scenario division module is similar to the definition of the use case which describes how users will perform the tasks, i.e., an outline of an application's behavior from a user's point of view. They describe the functions provided by the application without drilling into details. However, since different users have different understandings and a function may contain some sub-functions, the division of scenarios can be different. For example, the function of reading news can be a use case for a news app, but it can also be divided into two scenarios which include generally browsing a list of news and reading the details of one piece of news. However, this would not cause problems for our scenario division module. As we employ a neural network based learning framework, the granularity of dividing scenarios can be adjusted by feeding different training data.

Figure 3 shows three typical scenarios that are common in Android applications, and each scenario consists of two GUI states. As we can see, the states that are in the same scenario can be quite different and it will be difficult for existing state comparison criteria to make a correct division. Existing researches rely on manual defined features (e.g., executable events, GUI hierarchy, activity name) to describe the states. Some of them utilize a threshold to make the division flexible (e.g., two interfaces with 80% same widgets will be judged as the same state). However, prior approaches cannot meet our requirement where more properties should be considered and some of them are quite hard for a human to extract. Take Figure 3(a) as an example. There are two states in the browsing scenario which contains a RecyclerView to present items of news. These two states can access each other by scrolling. In order to successfully determine that the two states are in the same scenario, firstly, the number of items should be ignored. Second, the items' GUI hierarchies should not be viewed as equally important as other widgets. Despite the heavy work of feature selection, it is also a difficult task to manually figure out a probable threshold as it usually relies on a large amount of data and experiment. Q-testing tackles these problems under the help of the neural network which excels at extracting features.

### 4.2 Siamese LSTM

Q-testing utilizes the siamese network [8] to measure the similarity between two states from the perspective of judging whether they are in the same scenario. The siamese network is able to learn an invariant and selective representation through information about the similarity between sample pairs. It has been applied in NLP (Natural

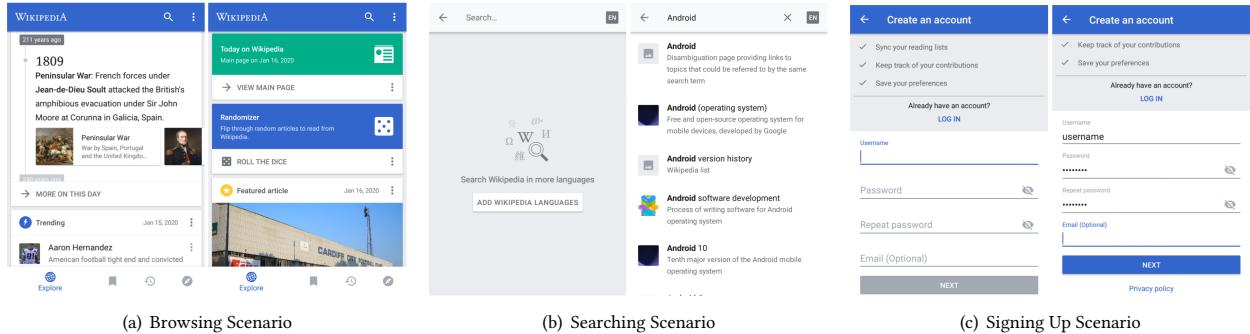


Figure 3: Examples of Scenarios in Android Applications

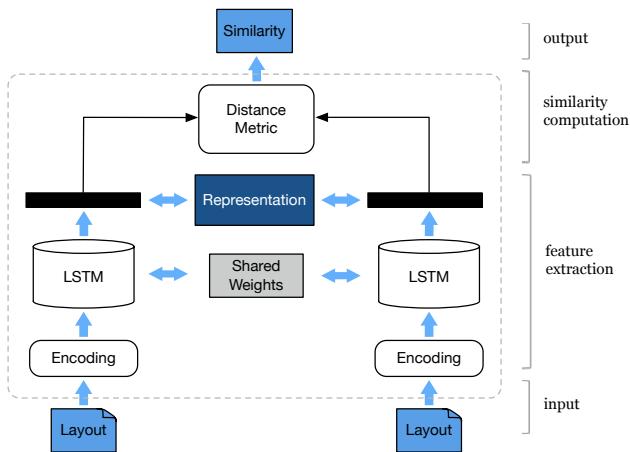


Figure 4: Architecture of Scenario Division Module

Language Processing) to learn the similarity between words [37] or sentences [36] whose goals share a lot in common with ours.

Figure 4 shows an overview of the network architecture in Q-testing. Each training sample of our task can be denoted as  $(s_a, s_b, y)$  where  $s_a$  and  $s_b$  are layout files dumped by UIAutomator. Label  $y \in \{0, 1\}$  depicts whether the two states can be divided into the same scenario ( $y = 1$ ) or not ( $y = 0$ ).

Layout files  $(s_a, s_b)$  which are in the form of XML will firstly be encoded as sequences  $(s_1^{(a)}, s_2^{(a)}, \dots, s_{l_a}^{(a)})$  and  $(s_1^{(b)}, s_2^{(b)}, \dots, s_{l_b}^{(b)})$  where  $s_i$  denotes the vector representation of a node in XML which contains information of a GUI widget. Q-testing utilizes different encoding strategies for different types of widget properties:

- **Integer.** The bounds attribute which contains 4 integers is used to demonstrate the position of a widget on the screen (2 integers to denote the position of the left top point and the other 2 denote the position of the right bottom point). Q-testing normalizes these values with the width and length of the screen.
- **Boolean.** The events related attributes (e.g., clickable, scrollable) often use boolean values to describe the executable

properties of a widget. Q-testing maps true and false into 1 and 0 to encode such attributes.

- **String.** Attributes such as resource id and text are represented in the form of string. When comparing two states, we only care about whether 2 strings are the same rather than what their actual values are. Q-testing hashes the texts into numbers with the MD5 message-digest algorithm.
- **String (Class Type).** The class attribute describes the widget's class type and it is in the form of string too. Q-testing processes this attribute with one-hot encoding since all the types of Android widgets can be accessed.

The encoded sequences will then be passed to the dual-LSTM (Long Short-Term Memory) network which comprises two single-layer networks with 100 hidden neurons. The neural network is able to learn a mapping from the space of variable length sequences into a fixed length vector. Finally, the distance of the generated vectors will be measured with similarity functions like Manhattan distance and the weights of the layers in LSTMs will be updated according to the difference between output and actual label. Note that the LSTMs share the same weights, they will extract features with the same standard.

After the training process, part of the model is used in the scenario division module. Q-testing leverages one LSTM along with its encoding module to extract features from state files. The states stored in memory buffer are also in the form of a feature vector. In this way, the time spent in comparing states will decrease since there is no need to extract features for all the remembered states. It is worth mentioning that other tasks that need state comparison or states compression may also benefit from such a module.

One of the challenges of applying the deep learning approach is that a large amount of training data is needed. We address this challenge in the following section.

### 4.3 Training Data Collection

In order to improve the model's performance, a wealth of samples should be collected. We collect and label the samples in 4 steps.

**Instrumenting APKs.** We collect plenty of commercial apps and each of them contains abundant scenarios. Since most of them

are closed-source apps, we conduct research on the Android framework and utilize Soot [40] to instrument several methods and statements of the APKs to collect runtime information which will provide more information to facilitate the later labeling process. Specifically, the instrumented methods and statements are corresponding to the start and destroy of Activity, Fragment or Dialog and often make a huge impact on the display of interface along with changes of scenarios. The invocations of these instrumented methods and statements usually mean a switch of scenarios. As for APKs do not support instrumenting, we will manually label samples collected from them.

**Collecting Data.** We implement a tool to automatically collect sample data. It randomly explores the applications and every time it executes an event, the states before and after the transition will be stored as a sample pair.

**Augmenting Data.** There are issues in the automatically collected data among which the most important one is that it only includes pairs where the two states can transit to each other with one event. This restriction will result in the different distribution between training data and actual data since a new state has to be compared with all the states with the same activity in memory.

In order to solve the problem, we make an augmentation of the collected data. More concretely, we divide the collected data into several groups *w.r.t.* their activities. Then new pairs are generated by randomly mapping the states in the same activity.

**Labelling Data.** After data collection steps, we look into the screenshots, the collected runtime information of every state pair and decide a label for the pair of whether it belongs to the same scenario or not.

The labeled samples are fed to the neural network for training. We pre-trained a model with 6058 pairs of samples for Q-testing. More details are in Section 5.

## 5 EVALUATION

In this section, we mainly inspect two parts: (1) the ability of the scenario division module; and (2) the ability of the whole testing tool to automatically covering codes and triggering bugs. We aim to answer the following research questions in our evaluation:

**RQ1: Scenario Division.** Is Q-testing able to determine different functional scenarios effectively?

**RQ2: Code Coverage.** How does Q-testing compare against state-of-the-art and state-of-practice testing tools with respect to code coverage?

**RQ3: Fault Revelation.** How does Q-testing compare against state-of-the-art and state-of-practice testing tools with respect to fault revelation?

### 5.1 Evaluation Setup

All the experiments are conducted on a physical machine with 4 cores 3.60GHz CPU and 16GB RAM on Ubuntu 16.04. Since the original Stoat [46] and Sapienz [33] are suggested to running on Android API level 19, We run all experiments on Android emulators which are configured with 2GB RAM, and the KitKat version (SDK 4.4, API level 19).

**Scenario Division.** We collect and label samples from 92 commercial Android apps with the approach described in Section 4.3. The automatic collecting tool explores each of the apps for at least one hour. Most of these popular apps comprise abundant scenarios that will benefit the classification accuracy of the neural network when applied to other applications. We conduct 5 rounds of 10-fold cross-validation. The division of data is conducted by apps that scenarios from the same application should not exist in both the training set and the testing set; otherwise, it may increase the classification accuracy but jeopardize the validation result.

**Automated Testing.** In this part of the evaluation, Q-testing is compared with Monkey [21], Stoat [46], and Sapienz [33], which are considered as the state-of-practice and state-of-the-art tools in Android GUI testing. Our benchmark consists of 50 open-source Android applications which are collected from two sources:

(1) Most of the applications are collected from related work [11, 41]. Apps selected by Choudhary et al. [10] have become a standard benchmark in evaluating automated testing tools and is adapted by several works [27, 33, 46]. Since many of these applications are quite out of date, we only involve those can be found in F-Droid [29] or GitHub [16]. We also exclude the projects that cannot be compiled due to long time no maintenance. After the filtering, 34 applications are selected.

(2) Over 40 percent of the applications selected above have less than 1k executable lines of codes (ELOC) which may be unable to reflect the difference between diversity exploration strategies. So we enrich the benchmark by adding 16 larger apps from the open-source apps list [39]. The applications are randomly selected from several commonly used categories (i.e., communication, news, education, tools) and most of them have more than 10k ELOC.

Additionally, applications that have been used to train scenario division module and gaming applications are excluded. Note that although the collected applications all have source-code, Q-testing can test application APKs without source-code. We use these open-source applications so that we can collect code coverage more accurately, analyze the root causes of faults with reference to the code, and better understand the testing behavior.

The testing time is set to one hour. We follow previous work [11, 24, 34] to set a 200 milliseconds wait interval for Monkey to partially avoid some abnormal behavior. Stoat's default time settings for 2 phases are one hour and two hours. We did not follow the allocation proportion as [46] in our one hour testing because we were not sure whether 20 minutes of modeling was enough. We conducted a small evaluation on several applications to compare the allocation of 40/20, 30/30, and 20/40 minutes for Stoat. The difference is not huge but the 30/30 minutes allocation does exceed. So all the apps are tested with this time allocation. As for Sapienz, we notice the evolution step can not be executed in some apps because of the large default setting for population size. So we run Sapienz with different population sizes for each app and pick the one with the highest code coverage. The selected setting is listed in Table 1. To mitigate randomness, for every testing tool, four test runs were conducted on the benchmark apps. We also enable Stoat and Sapienz to compute code coverage with Jacoco [14] for consistency.

## 5.2 Experimental Results

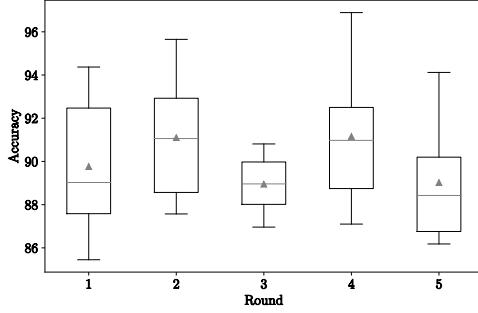


Figure 5: Accuracy of Scenario Division

**RQ1: Scenario Division.** We totally collect and label 6058 pairs of samples. For every round of 10-fold cross-validation, the samples are divided into 10 sets and every set is selected as the testing set by turns. The neural network is trained with approximately 90 percent of the data for 100 epochs which costs about half an hour on our experiment PC. After that, the trained model is applied to the testing set to complete a test. After 10 repeated operations, the average accuracy is calculated. Figure 5 shows the result of the 5 rounds of 10-fold cross-validation. The average accuracy is 89.80% and the lowest accuracy is above 85%, which indicates the model's ability to divide different scenarios.

We also undertake a careful analysis of the classification results. The examples in Figure 3, which are important scenarios in Android applications, are all correctly identified as the same scenario by our model. These are also samples difficult for previous criterion [6], including executable events and widget layout, to make a correct judgment.

As for those predicted mistakenly by our model, some of them are difficult to distinguish even by human users. Since the design of Android applications is quite flexible, it is sometimes hard to clearly define whether 2 states belong to the same scenario. Take Figure 6 as an example. The left state has a search bar on the top of several pieces of news which makes it look like a combination of searching scenario and browsing scenario. Even if the right state can be reached by simply scrolling down in the left state, it is hard to category the two states into the same scenario. Fortunately, our reinforcement learning framework is able to tolerate mistakes made by the scenario division module because of the bellman function. On the one hand, if the module makes a mistake and updates the event's value wrongly with a large number, its value will still be rectified later after several triggering and updating due to value propagation. On the other hand, if a new scenario which leads to important states is reached and the neural network mistakenly updates the event's value with a small number, the executed event's value will still be updated larger by the bellman function. With this function, all the subsequent events' value will make an effect on the previous one.

In conclusion, it is reliable to determine different scenarios and calculate reinforcement learning rewards with the scenario division module.

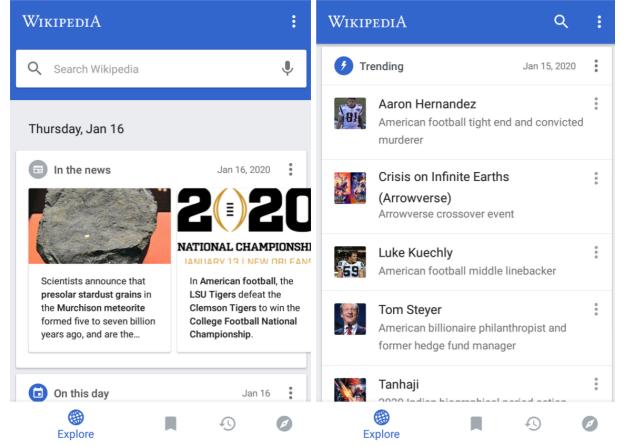


Figure 6: Example of Confusing Scenario

**RQ2: Code Coverage.** In this experiment, Q-testing uses the siamese network trained with the 6058 pairs of samples to divide scenarios.

Table 1 shows the average instruction coverage of the four test runs by Monkey, Stoat, Sapienz, and Q-testing on 50 open source applications. The apps are sorted by ELOC extracted from Jacoco coverage report and the first two apps are too small for Stoat to conduct the second phase. On average, Q-testing achieves 46.62% instruction coverage which is higher than Monkey (43.50%), Stoat (38.82%) and Sapienz (40.48%). Additionally, Q-testing achieves the highest code coverage for 33 of the open-source applications while the statistics for Monkey, Stoat, and Sapienz are 8, 7 and 9.

Coverage information is collected every 30 seconds during testing and Figure 7 depicts the progressive average code coverage for each tool in one hour. Monkey achieves the highest coverage within the first few minutes. That is mainly because it executes events at an extraordinarily high speed and lots of random events can lead it to new states in the very beginning. However, as the progress of testing continues, more and more redundant events are executed by Monkey and its efficiency starts to decrease. Q-testing takes the first place after about 5 minutes which confirms the effectiveness of our curiosity-driven strategy. The strategy to firstly discover different scenarios enables Q-testing to achieve high code coverage within a short time. This is extremely important when the testing budget is tight.

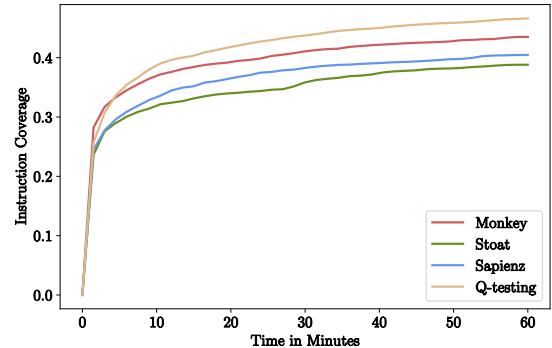
The efficiency of generating events is also important when testing time is limited. During one hour's testing, Monkey, Stoat, Sapienz, and Q-testing averagely generate approximately 53000, 1804, 29570 and 1693 events. As for Monkey, there is no need for it to analyze widget hierarchy before generating events. Besides, it can communicate directly with several essential Android services. These features enable it to generate events with high speed. Even if Monkey can generate much more events than others, most of them are redundant and may make no effect on the applications. Sapienz comes second in generating events as it is built upon Monkey. It is encouraging to see the number of executed events by Q-testing is close to that of Stoat's. Stoat can directly generate event sequences with reference to models, which benefits the testing efficiency. Q-testing does not

**Table 1: Testing Results for Comparison**

Subject	Name	ELOC	Coverage(%)				Faults				Setting Sa
			M	St	Sa	Q	M	St	Sa	Q	
DivideAndConquer		80	91	-	95	95	0	-	0	0	50
QuickSettings		91	91	-	92	91	1	-	1	1	20
MunchLife		188	81	86	82	90	0	0	0	0	50
AnyCut		414	66	58	64	67	0	0	0	0	20
lockpatterngenerator		674	83	68	83	65	0	0	0	0	20
autoanswer		447	15	18	16	21	2	2	0	2	30
batterydog		468	59	51	60	49	0	0	1	1	50
Dumbphone		572	40	38	35	40	0	0	0	1	30
soundboard		641	38	44	32	46	3	0	0	1	30
whohasmystuff		744	77	75	68	78	1	1	0	2	50
zooborns		760	17	19	16	17	1	3	1	4	20
multimssender		862	55	55	59	45	3	1	0	0	50
alogcat		906	71	67	71	78	0	0	0	0	20
SmsScheduler		915	58	61	52	57	0	0	0	0	30
dialer		955	45	45	47	49	1	3	2	2	20
Talaraldo		1030	74	78	74	77	1	0	0	1	30
WeightChart		1054	67	58	67	78	6	2	3	2	50
alarmclock		1269	43	47	41	36	1	4	1	2	20
Notes		1910	59	55	54	73	3	0	3	2	20
PasswordMaker		1949	60	55	57	61	0	2	0	5	50
BudgetWatch		2143	42	45	47	61	1	0	1	6	40
manpages		2417	18	12	15	17	2	5	2	3	30
GoodWeather		2501	69	58	57	73	4	4	0	5	20
swiftp		3188	21	23	22	22	1	3	1	5	30
jammedo		3904	29	14	41	46	3	4	2	6	30
fillup		3967	58	57	43	60	1	5	0	1	40
sanity		4675	24	15	23	26	3	1	2	5	20
mileage		4711	44	35	45	42	2	8	2	7	20
importcontacts		4817	2	2	2	2	1	1	1	1	40
Tomdroid		4901	42	49	46	50	1	0	0	2	20
materialistic		6661	54	46	26	54	2	1	0	5	40
RadioBeacon		7459	35	39	32	41	4	4	1	11	20
TimBrowser		7575	38	40	16	43	0	4	0	0	30
AntennaPod		8153	43	34	24	42	9	2	0	9	30
keepassdroid		9035	11	8	11	8	0	0	0	2	20
ConnectBot		9090	21	23	16	26	2	1	0	5	50
APhotoManager		9751	50	36	46	55	3	1	6	5	50
BetterBatteryStats		10042	11	16	15	18	1	4	0	6	50
uhabits		10629	55	45	36	54	3	4	4	3	20
vanilla		10776	41	46	35	40	6	1	4	4	40
Timber		12004	43	30	31	36	9	6	4	10	40
AnyMemo		12414	31	20	44	46	6	15	2	10	30
Runnerup		16378	19	20	22	23	11	9	8	11	40
SuntimesWidget		16681	40	38	16	48	1	2	0	6	40
AmazeFileManager		17705	26	26	27	29	4	6	4	6	30
amme		21586	17	12	15	30	3	2	3	9	20
BookCatalogue		24378	28	30	29	35	2	1	0	6	50
Anki		28093	29	19	33	32	0	6	3	13	50
MyExpenses		29067	25	27	26	34	0	4	0	5	50
Signal		43954	21	20	20	26	1	8	0	4	30

have a model to guide the generation of test sequences. However, the Q-table acts like a lightweight model which maps explored states to actions and can save the time inferring executable events. What's more, the storage of states' vector rather than the original GUI hierarchy in memory set contributes to the improvement of efficiency too.

We also dissect the codes covered by different tools. Take GoodWeather which is a weather application as an example. In GoodWeather, a considerable portion of functions can be configured for variants via a ‘Setting’ option residing in a Sidebar. Q-testing outperforms other tools on GoodWeather mainly because it tries more settings during testing. Some of the settings have to be executed with at least 6 steps and it is a long event sequence for Android applications. There are a lot of candidate events in every step which

**Figure 7: Progressive Instruction Coverage**

makes the settings hard to trigger. To be more specific, we make an analysis of the testing behavior of Q-testing and discover that Q-testing tends to open the Sidebar (with 60.77% probability) in the main interface which is not viewed in Stoat (14.15%). Monkey and Sapienz do have the same tendency to open the Sidebar. That is because they would try to execute keyboard events and tend to click the button on the top left of the interface. However, keyboard events are not available in current mobiles, and clicking the top left area is now to open the Sidebar in GoodWeather. This phenomenon can also be viewed when the Sidebar is opened where Monkey would choose the first option with a greater possibility (43.20%) than the fourth ‘Settings’ option (13.02%). In Sidebar, where there are 6 candidate options and several other events (e.g. back, menu) to be chosen, Q-testing selects the ‘Settings’ option with a probability of 30.69%. This observed phenomenon indicates the power of curiosity-driven strategy and the value propagation property of Q-learning for exploring important scenarios which can lead to more new scenarios.

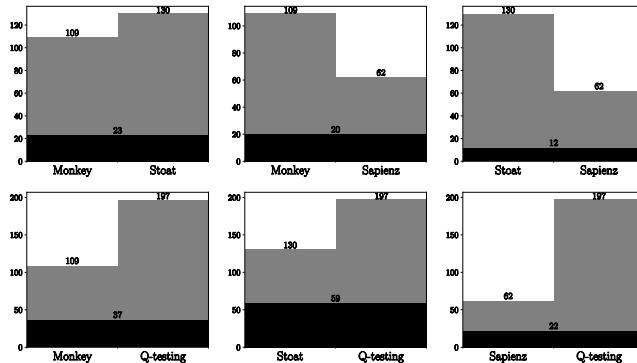
**RQ3: Fault Revelation.** Table 1 also shows the unique faults revealed by the four testing tools. We adopt the definition in Stoat where a fault is identified by crash or exception lines in stack traces (exceptions without the keywords of the AUT’s package name are excluded). Q-testing detects the most faults for 27 of the experiment applications (tied first also counts) while Stoat detects the most for 12 of them and comes second. Q-testing reveals a total of 197 faults among the 50 applications, which is more effective than Monkey (109), Stoat (130), and Sapienz (62).

Figure 8 shows the pairwise comparison result of revealed faults between each 2 of the testing tools. The intersecting faults found by Stoat and Q-testing are 59 and most of them are triggered by system events. More than 30% of faults revealed by Sapienz is covered by Monkey since they share the same manner in injecting events. In addition, the numbers of unique faults, which are not revealed by the other 3 tools, for Q-testing, Monkey, Stoat, and Sapienz are 115, 63, 67, and 32. This shows the ability of Q-testing to trigger faults, which can not be replaced by other tools.

Most of the faults revealed by Q-testing have been reproduced and reported to the developers. So far, 22 reported issues have been confirmed to be first-time found real faults. We consider an issue confirmed if the developers respond in text clearly or add a ‘Bug’

**Table 2: Confirmed Issues Found by Q-testing**

App Name	Exception	Description	Status	Issue URL
Anki-Android	ActivityNotFoundException	No Activity found to handle a View Intent	Fixed	<a href="https://github.com/ankidroid/Anki-Android/issues/5653">github.com/ankidroid/Anki-Android/issues/5653</a>
SuntimesWidget	NullPointerException	Multi thread related crash	Fixed	<a href="https://github.com/forrestguice/SuntimesWidget/issues/376">github.com/forrestguice/SuntimesWidget/issues/376</a>
Runnerup	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Fixed	<a href="https://github.com/jonasoreland/runnerup/issues/862">github.com/jonasoreland/runnerup/issues/862</a>
Runnerup	IllegalStateException	Add a child view which already has a parent	Fixed	<a href="https://github.com/jonasoreland/runnerup/issues/863">github.com/jonasoreland/runnerup/issues/863</a>
Book-Catalogue	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	<a href="https://github.com/eleybourn/Book-Catalogue/issues/835">github.com/eleybourn/Book-Catalogue/issues/835</a>
manpages	InflateException	Error inflating class on binary XML file	Confirmed	<a href="https://github.com/Adonai/Man-Man/issues/19">github.com/Adonai/Man-Man/issues/19</a>
manpages	NullPointerException	Crash after cache is cleared	Confirmed	<a href="https://github.com/Adonai/Man-Man/issues/20">github.com/Adonai/Man-Man/issues/20</a>
Swiftfp	NullPointerException	Unable to start FsWidgetProvider Receiver	Confirmed	<a href="https://github.com/ppareit/swiftfp/issues/140">github.com/ppareit/swiftfp/issues/140</a>
Swiftfp	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	<a href="https://github.com/ppareit/swiftfp/issues/150">github.com/ppareit/swiftfp/issues/150</a>
Budget-Watch	WindowsLeakedException	Dialog's state not preserved on screen orientation or activity close	Confirmed	<a href="https://github.com;bracher/budget-watch/issues/202">github.com;bracher/budget-watch/issues/202</a>
Budget-Watch	WindowsLeakedException	TransactionActivity has leaked window that was originally added	Confirmed	<a href="https://github.com;bracher/budget-watch/issues/203">github.com;bracher/budget-watch/issues/203</a>
Budget-Watch	NullPointerException	Unable to start ReceiptViewActivity	Confirmed	<a href="https://github.com;bracher/budget-watch/issues/204">github.com;bracher/budget-watch/issues/204</a>
Notes	IndexOutOfBoundsException	Cursor out of bounds when handling Intent in NotificationService	Confirmed	<a href="https://github.com/SecUsO/privacy-friendly-notes/issues/77">github.com/SecUsO/privacy-friendly-notes/issues/77</a>
Notes	IllegalArgumentException	Illegal argument in SketchActivity when editing reminder	Confirmed	<a href="https://github.com/SecUsO/privacy-friendly-notes/issues/78">github.com/SecUsO/privacy-friendly-notes/issues/78</a>
AmazeFileManager	IllegalStateException	Illegal state while executing doInBackground()	Confirmed	<a href="https://github.com/TeamAmaze/AmazeFileManager/issues/1793">github.com/TeamAmaze/AmazeFileManager/issues/1793</a>
AmazeFileManager	WindowsLeakedException	Dialog's state is not preserved on screen orientation	Confirmed	<a href="https://github.com/TeamAmaze/AmazeFileManager/issues/1794">github.com/TeamAmaze/AmazeFileManager/issues/1794</a>
AmazeFileManager	ActivityNotFoundException	Intent cannot be handled when amaze cloud plugin is not available	Fixed	<a href="https://github.com/TeamAmaze/AmazeFileManager/issues/1795">github.com/TeamAmaze/AmazeFileManager/issues/1795</a>
AmazeFileManager	IndexOutOfBoundsException	Index out of bounds when cutting a folder and pasting within itself	Confirmed	<a href="https://github.com/TeamAmaze/AmazeFileManager/issues/1796">github.com/TeamAmaze/AmazeFileManager/issues/1796</a>
AmazeFileManager	NullPointerException	Unable to start TextEditorActivity	Fixed	<a href="https://github.com/TeamAmaze/AmazeFileManager/issues/1808">github.com/TeamAmaze/AmazeFileManager/issues/1808</a>
Uhabits	NullPointerException	WeekdayPickerDialog's not preserved results in null object reference	Fixed	<a href="https://github.com/iSoron/uhabits/issues/534">github.com/iSoron/uhabits/issues/534</a>
PasswordMaker	IndexOutOfBoundsException	Index out of bounds when deleting folder	Confirmed	<a href="https://github.com/passwordmaker/android-passwordmaker/issues/47">github.com/passwordmaker/android-passwordmaker/issues/47</a>
AnyMemo	NullPointerException	Crash when trying to paint because the library used is not maintained	Confirmed	<a href="https://github.com/helloworld1/AnyMemo/issues/488">github.com/helloworld1/AnyMemo/issues/488</a>

**Figure 8: Pairwise Comparison on Fault Revelation**

label manually. Among these 22 faults, 7 have been fixed. Table 2 list the confirmed issues along with the bug types, brief description, and statuses. Several bugs are quite concealable. For example, one multi-thread related crash in AmazeFileManager can only be triggered when a zip file, which is contained in another zip file, is continuously selected to be extracted. The event traces recorded by Q-testing help a lot in reproducing these faults. Considering that most of the apps have been maintained for quite a while and tested many times by previous work, revealing such an amount of new faults clearly demonstrates Q-testing’s ability.

### 5.3 Threats to Validity

**Internal Threats.** The main internal threat lies in the choice of parameter settings for the four testing tools that may affect the testing results. In order to mitigate the threat, we try to choose their default settings as possible. For circumstances where default settings cannot be adopted, we conduct small-scale experiments and choose suitable settings before the formal evaluation. We cannot figure out a good setting for Sapienz’s population size, so we run it

with different settings and record the one that has the best testing effect.

The performance of siamese network is also an internal threat to validity. As for this threat, we conduct experiments to verify its ability. We will collect more training data and optimize the model in further work.

**External Threats.** The external threat mainly lies in the selection of subject apps and our results may not be generalized to other apps. We alleviate the threat by choosing subject apps from related work which include a standard benchmark. We further refine the benchmark by replacing out of date apps with large-scale ones from a popular open-source list.

## 6 RELATED WORK

Many techniques have been proposed to automate Android GUI testing. We broadly classify the technologies into four categories w.r.t. their exploration strategies.

**Random Testing.** This kind of testing tools [21, 31] adopt random strategies to generate input for Android applications. Monkey [21], which is the most frequently used Android testing tools, generates pseudo-random streams of user events by randomly interacting with screen coordinates. This basic random strategy performs quite well on some benchmark apps [11]. However, the generated test cases contain a large number of non-effective or redundant events and this will be a threat to the testing effectiveness.

Some other tools [22, 42, 54] utilize fuzzing testing to generate intent inputs rather than explore applications’ states to make the AUT crash or to reveal security issues. Q-testing also takes intent inputs into consideration when generating system-level events.

**Model Based Testing.** Model-based approaches [2, 3, 5, 6, 13, 23, 27, 44, 51–53] build models with dynamic or static strategies to describe the applications’ behaviors and then derive test cases from the models to find bugs. Since the test cases are generated underlying the constructed model, its accuracy and completeness will be of great importance.

Stoat [46] utilizes a stochastic Finite State Machine model to describe the behavior of AUT. In the model construction process, it infers the executable events and prioritizes their executions according to factors like event type and executed frequency. Then in the test generation process, Stoat leverages MCMC sampling to direct the mutation of the model from which test cases are derived.

Q-testing leverages Q-table to record executable events in the covered state and the transition information is partially considered by the Bellman function. To some extent, Q-testing also benefits from model-based testing. However, Q-testing does not directly derive test cases from a model, which prevents it from the problem of inconsistent between model and applications' actual behavior. APE [24] makes some effort in model abstraction and refinement which alleviates the problem, but the model is still imprecise as internal states of variables, which will affect the applications' behavior, are still ignored due to limitation of the modeling language.

**Systematic Testing.** Symbolic execution and evolutionary algorithms are applied by systematic strategies [4, 15, 32] to generate specific input to cover hard-to-reach codes.

Sapienz [33] leverages a Pareto-optimal multi-objective search-based approach to maximize code coverage and bug revelation while at the same time minimizing the length of test sequences. It also reverses-engineering the APK to get statically-defined strings to generate specific input for text fields. Sapienz generates new test cases by random crossover and mutation, which will result in invalid sequences. The iterative evaluation of new generated test cases will also cost a lot of time.

**Machine Learning Based Testing.** Machine learning has been applied in Android GUI testing and related work can be divided into 2 categories. The first kind of approaches [7, 26, 28, 30] have an explicit training process to learn from the previous testing process and the learned experience will later be leveraged on new apps. QBE [26] learns from a set of Android applications the value of different types of events towards a particular objective like increasing activity coverage or detecting crashes. In order to make it possible for knowledge to be transferred between different applications, these approaches usually have to make abstractions on the apps' states. QBE divides states by the number of enabled actions where plenty of information is ignored and this may reduce testing effectiveness. Additionally, the design of Android applications is quite flexible, it may be useless to guide the testing of new applications with knowledge learned from others.

The other work tends to model independently for every app [1, 47, 48] or adapts a general model to the app under test [12]. Some of them [1, 47, 48] extend AutoBlackTest [35] and do not have an explicit training process. Similar to Q-testing, they also apply Q-learning to guide the exploration of Android applications. However, they simply rely on the difference between two states' executable events and their executing frequency to determine reward where the events' value can not be adjusted flexibly during testing. For example, an event connecting two states with quite different events will always be encouraged to execute when most events are executed many times. Q-testing tackles this problem by using memory and scenario division module to steer the testing towards unfamiliar functionalities with high efficiency.

Reinforcement learning is also applied to other testing work. For example, Wuji [55] combines reinforcement learning with evolutionary algorithms to test games. Retecs [45] uses RL to guide test prioritization and selection in regression testing.

## 7 CONCLUSION

In this paper, we propose Q-testing, a Q-learning based approach for Android automated testing. Q-testing leverages Q-table as a lightweight model while exploring unfamiliar functionalities with a curiosity-driven strategy. To effectively determine the reward for Q-learning and to further guide the exploration, a scenario division module which distinguishes functional scenarios using a neural network is proposed. Experiments show that Q-testing outperforms the state-of-the-art and state-of-practice Android GUI testing tools in both code covering and fault detection.

## ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program (Grant No. 2017YFB1001801), the National Natural Science Foundation (Nos. 61632015, 61972193), and the Fundamental Research Funds for the Central Universities (Nos. 14380022, 14380020) of China.

## REFERENCES

- [1] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement learning for Android GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 2–8.
- [2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 258–261.
- [3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.
- [4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 1–11.
- [5] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Acm Sigplan Notices*, Vol. 48. ACM, 641–660.
- [6] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *2016 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 238–249.
- [7] Nataniel P Borges Jr, María Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 133–143.
- [8] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a "siamese" time delay neural network. In *Advances in neural information processing systems*. 737–744.
- [9] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A Efros. 2019. Large-scale study of curiosity-driven learning. In *7th International Conference on Learning Representations (ICLR)*.
- [10] Shaumik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.
- [11] Shaumik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.
- [12] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 296–306.
- [13] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd*

- IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007.* ACM, 31–36.
- [14] EclEmma. 2020. JaCoCo Java Code Coverage Library. <https://www.eclemma.org/jacoco/index.html>
- [15] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *2018 Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 419–429.
- [16] GitHub. 2020. GitHub. <https://github.com/>
- [17] Google. 2019. Broadcasts overview. <https://developer.android.google.cn/guide/components/broadcasts>
- [18] Google. 2019. ListView. <https://developer.android.google.cn/reference/android/widget/ListView>
- [19] Google. 2019. Recyclerview. <https://developer.android.google.cn/reference/androidx/recyclerview/widget/RecyclerView>
- [20] Google. 2019. UI Automator. <https://developer.android.com/training/testing/ui-automator>
- [21] Google. 2019. UI/Application Exerciser Monkey. <https://developer.android.com/studio/test/monkey>
- [22] NCC group. 2012. Intent Fuzzer. <https://www.nccgroup.trust/us/our-research/intent-fuzzer/>
- [23] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Liu. 2017. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSM)*. IEEE, 103–114.
- [24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 269–280.
- [25] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. 2016. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*. 1109–1117.
- [26] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanrıverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 105–115.
- [27] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 115–127.
- [28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.
- [29] F-Droid Limited. 2020. F-Droid - Free and Open Source Android App Repository. <https://f-droid.org/>
- [30] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53.
- [31] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 224–234.
- [32] Riyad Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 599–609.
- [33] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.
- [34] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 16–26.
- [35] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. 2012. Autoblocktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 81–90.
- [36] Jonas Mueller and Aditya Thyagarajan. 2016. Siamese Recurrent Architectures for Learning Sentence Similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) (AAAI'16). AAAI Press, 2786–2792.
- [37] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. 2016. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*. 148–157.
- [38] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. 2017. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*. 2778–2787.
- [39] pcqpcq. 2020. Open-Source Android Apps. <https://github.com/pcqpcq/open-source-android-apps>
- [40] Sable. 2019. Soot - A framework for analyzing and transforming Java and Android applications. <https://sable.github.io/soot/>
- [41] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. Padtdroid: permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 220–232.
- [42] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 1–5.
- [43] Nikolay Savinov, Anton Raichuk, Raphaël Marinier, Damien Vincent, Marc Pollefeyns, Timothy Lillicrap, and Sylvain Gelly. 2019. Episodic curiosity through reachability. In *7th International Conference on Learning Representations (ICLR)*.
- [44] M Shafique and Y Labiche. 2010. A systematic review of model based testing tool support, Technical Report SCE-10-04. Carleton University, Canada (2010).
- [45] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 12–22.
- [46] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.
- [47] Tuyet Vuong and Shingo Takada. 2019. Semantic analysis for deep Q-network in android GUI testing. In *31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*. Knowledge Systems Institute Graduate School, 123–128.
- [48] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A reinforcement learning based approach to automated testing of Android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, 31–37.
- [49] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3–4 (1992), 279–292.
- [50] Yoram Wurmser. 2018. Mobile Time Spent 2018. <https://www.emarketer.com/content/mobile-time-spent-2018>
- [51] Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang. 2018. LAND: a user-friendly and customizable test generation tool for Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 360–363.
- [52] Jiwei Yan, Tianyong Wu, Jun Yan, and Jian Zhang. 2017. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 42–53.
- [53] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.
- [54] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. ACM, 68.
- [55] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yinfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 772–784.



# Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy

Seokhyun Lee  
Korea University  
Republic of Korea  
seokhyunlee@korea.ac.kr

Sooyoung Cha  
Korea University  
Republic of Korea  
sooyoungcha@korea.ac.kr

Dain Lee  
Korea University  
Republic of Korea  
dain\_lee@korea.ac.kr

Hakjoo Oh\*  
Korea University  
Republic of Korea  
hakjoo\_oh@korea.ac.kr

## ABSTRACT

We present ADAPT, a new white-box testing technique for deep neural networks. As deep neural networks are increasingly used in safety-first applications, testing their behavior systematically has become a critical problem. Accordingly, various testing techniques for deep neural networks have been proposed in recent years. However, neural network testing is still at an early stage and existing techniques are not yet sufficiently effective. In this paper, we aim to advance this field, in particular white-box testing approaches for neural networks, by identifying and addressing a key limitation of existing state-of-the-arts. We observe that the so-called neuron-selection strategy is a critical component of white-box testing and propose a new technique that effectively employs the strategy by continuously adapting it to the ongoing testing process. Experiments with real-world network models and datasets show that ADAPT is remarkably more effective than existing testing techniques in terms of coverage and adversarial inputs found.

## CCS CONCEPTS

- Computer systems organization → Neural networks;
- Software and its engineering → Software testing and debugging

## KEYWORDS

Deep neural networks, White-box testing, Online learning

### ACM Reference Format:

Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective White-Box Testing of Deep Neural Networks with Adaptive Neuron-Selection Strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397346>

## 1 INTRODUCTION

Testing deep neural networks is becoming increasingly important. Nowadays, deep neural networks are pervasive in many application domains, including image captioning [25, 36], game playing [28], and safety-critical domains such as medical diagnosis [26] and

\*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397346>

self-driving cars [2]. Since these applications include deep neural networks as critical components, ensuring that neural networks behave as expected has become a pressing issue. In response, recent years have seen a surge of interest in techniques for testing deep neural networks and identifying their defects before deployment [12, 18, 22, 24, 31, 32, 34]. Furthermore, because traditional coverage metrics (e.g. branch coverage) are not suitable for measuring the effectiveness of neural-network testing [24], developing specialized coverage criteria for deep neural networks continues to be another active research area [15, 18, 24, 27, 32].

Existing testing techniques for deep neural networks are broadly classified into grey-box [22, 34] and white-box approaches [12, 24]. Grey-box testing techniques are based on the idea of coverage-guided fuzzing [37] that has been popular for traditional software, where neural networks are instrumented to trace coverage information and this information is used to generate new test cases that are likely to increase coverage. On the other hand, white-box testing techniques exploit the internals of neural networks more aggressively as follows: (1) they select the internal neurons, (2) calculate the gradients of the outputs of the selected ones (with respect to the input), and (3) generate new test cases by adding the gradients to the original test case in the direction of increasing the output values. Compared to grey-box testing, white-box approaches are therefore likely to achieve higher coverage and find more defects.

A key ingredient of existing white-box approaches is namely a neuron-selection strategy. To determine the direction of the mutation, white-box approaches first select a set of internal neurons and then calculate their gradients. Thus, the ultimate effectiveness of white-box testing depends on the selection of the neurons in the first step. A number of heuristic neuron-selection strategies have been proposed to maximize coverage in a limited time budget. For instance, DeepXplore [24] uses a strategy that randomly selects unactivated neurons. DLFuzz [12] proposed four different neuron-selection strategies for prioritizing neurons covered frequently/rarely, neurons with top weights, and neurons near the activation threshold.

In this paper, we present a new white-box testing technique for deep neural networks. Our technique differs from previous white-box techniques in a crucial way. Existing white-box techniques use the gradients of a select set of internal neurons but the selection is done by a predetermined strategy. However, in this paper, we show that using a fixed neuron-selection strategy is a major limitation of the existing white-box approaches; specifically, we observed that the performance of the existing approaches is not consistent across diverse neural network models and coverage metrics. For instance, although a state-of-the-art tool, DLFuzz, performs well for testing LeNet-5 [17] with Top- $k$  Neuron Coverage (TKNC) [18],

we found that DLFuzz becomes inferior even to a naive random approach for ResNet-50 [13] with TKNC. In this paper, instead of using a predetermined strategy, we present a new testing technique that is able to adaptively determine neuron-selection strategies during the testing process. To do so, we propose a parameterized neuron-selection strategy and provide an online learning algorithm to adjust its parameters effectively.

Experimental results show that our new proposal remarkably improves the effectiveness of white-box testing for neural networks. We implemented the technique in a tool called ADAPT and compared its performance with five existing testing techniques with two coverage metrics, two datasets, and four real-world neural networks. In all experimental settings, ADAPT achieved consistently higher coverage and found more adversarial inputs than existing techniques. On average, ADAPT was 3 times more effective in increasing coverage than other techniques and 6 times more effective in finding various adversarial inputs.

**Contributions.** Our contributions are as follow:

- We present a new white-box testing approach for deep neural networks. The key novelty is to combine white-box testing with an algorithm that adapts the neuron-selection strategy.
- We demonstrate the effectiveness of our approach with five existing techniques and various experimental settings.
- We provide our implementation as an open-source tool. All experimental results are reproducible.<sup>1</sup>

## 2 PRELIMINARIES

In this section, we provide background on neural networks, coverage metrics, and existing white-box testing approaches.

### 2.1 Deep Neural Networks

A deep neural network (*DNN*) is a collection of layers:

$$DNN = \{L_1, L_2, \dots, L_D\}$$

where  $D$  denotes the number of layers called depth. We call  $L_1$  the input layer,  $L_D$  the output layer, and  $L_2, \dots, L_{D-1}$  the hidden layers. We define a layer  $L_d \in DNN$  to be a set of neurons:

$$L_d = \{n_{d,1}, n_{d,2}, \dots, n_{d,t_d}\}$$

where  $t_d$  indicates the number of neurons in layer  $L_d$ . Each neuron in layer  $L_d$  ( $2 \leq d \leq D$ ) is fully connected<sup>2</sup> to the preceding layer  $L_{d-1}$ , where each connection between  $n_{d-1,i}$  and  $n_{d,j}$  is associated with a weight  $w_{d,i,j} \in \mathbb{R}$ , where  $\mathbb{R}$  denotes real numbers. The weights for neurons in layer  $L_d$  ( $2 \leq d \leq D$ ) can be represented by a  $t_{d-1} \times t_d$ -matrix  $W_d$  as follows:

$$W_d = \begin{bmatrix} w_{d,1,1} & w_{d,1,2} & \cdots & w_{d,1,t_d} \\ w_{d,2,1} & w_{d,2,2} & \cdots & w_{d,2,t_d} \\ \vdots & & \ddots & \vdots \\ w_{d,t_{d-1},1} & w_{d,t_{d-1},2} & \cdots & w_{d,t_{d-1},t_d} \end{bmatrix}$$

<sup>1</sup><https://github.com/kupl/ADAPT>

<sup>2</sup>For presentation simplicity, we focus on fully-connected neural networks but our technique is also applicable to convolutional neural networks.

Assume a  $t_1$ -dimensional input vector  $I \in \mathbb{R}^{t_1 \times 1}$  is given. We can “execute” the neural network with the input  $I$  as follows:

$$V_1 = I, \quad V_d = \text{Layer}_d(V_{d-1}) \quad (2 \leq d \leq D)$$

where  $V_d = \langle v_{d,1}, \dots, v_{d,t_d} \rangle^T$  denotes the output of layer  $L_d$  and  $v_{d,i}$  the output of neuron  $n_{d,i}$ .  $\text{Layer}_d$  is a layer function that takes the output  $V_{d-1}$  of the previous layer and produces the output for the layer  $L_d$  as follows:

$$\text{Layer}_d(V_{d-1}) = \text{Activation}_d(W_d^T V_{d-1})$$

where  $\text{Activation}_d$  is a non-linear activation function for layer  $L_d$ . For instance, the dominant ReLU (Rectified Linear Unit) activation function [21] is defined as follows:

$$\text{Activation}_d(\langle x_1, \dots, x_{t_d} \rangle^T) = \langle \max(x_1, 0), \dots, \max(x_{t_d}, 0) \rangle^T$$

Given a neural network *DNN* and an input vector  $I \in \mathbb{R}^{t_1 \times 1}$ , let  $\text{Run}(DNN, I)$  be the set of output values of hidden layers in *DNN* when *DNN* is executed with  $I$ :

$$\text{Run}(DNN, I) = \{V_2, V_3, \dots, V_{D-1}\}.$$

Note that  $V_d$  denotes the output of all neurons in layer  $L_d$ . Thus,  $\text{Run}(DNN, I)$  describes the full internal state of *DNN*.

### 2.2 Coverage Metrics

The adequacy and effectiveness of software testing are typically measured by coverage metrics (coverage criteria). Recently, various coverage metrics for neural networks have been proposed [15, 18, 24, 32]. In this paper, we evaluate and compare testing techniques with two metrics: Neuron Coverage (NC) [24] and Top- $k$  Neuron Coverage (TKNC) [18].

We define a coverage metric, denoted Cov, as a function from the outcome of Run to a set of coverage identifiers. For example, the Neuron Coverage (NC) metric uses the locations of neurons as identifiers (the location of neuron  $n_{d,i}$  is  $(d, i)$ ). NC identifies the neurons whose output values are greater than a certain threshold  $\theta$ . The function Cov for NC is defined as follows:

$$\begin{aligned} \text{Cov}(\{V_2, \dots, V_{D-1}\}) \\ = \{(d, i) \mid v_{d,i} > \theta \wedge v_{d,i} \in V_d \wedge d \in \{2, \dots, D-1\}\}. \end{aligned}$$

The Top- $k$  Neuron Coverage (TKNC) metric is more fine-grained; it collects the  $k$  neurons with the highest output values in each layer. The coverage function Cov for TKNC is:

$$\begin{aligned} \text{Cov}(\{V_2, \dots, V_{D-1}\}) \\ = \bigcup_{d \in \{2, \dots, D-1\}} \{(d, i) \mid v_{d,i} \in \underset{S \subseteq V_d \wedge |S|=k}{\operatorname{argmax}} \sum_{s \in S} s\}. \end{aligned}$$

In both cases, the coverage ratio is calculated by  $\frac{|\text{Cov}(\{V_2, \dots, V_{D-1}\})|}{|L_2 \cup \dots \cup L_{D-1}|}$ . In the rest of this paper, we assume a coverage metric Cov is given.

### 2.3 White-Box Testing of Neural Networks

One of the main goals of neural network testing is to generate test cases (i.e. input vectors) that maximize a given coverage metric (Cov) in a limited testing budget. Assuming that the budget is given as the number of test cases, denoted  $n$ , the objective of testing

**Algorithm 1** White-box Testing for Neural Networks

---

```

1: procedure TESTINGDNN(DNN, I, Cov)
2:   C  $\leftarrow$  Cov(Run(DNN, I))
3:   repeat
4:     W  $\leftarrow$  {I}
5:     while W  $\neq \emptyset$  do
6:       I'  $\leftarrow$  Pick an input from W
7:       W  $\leftarrow$  W \ {I'}
8:       N  $\leftarrow$  Strategy(DNN)
9:       for t = 1 to  $\eta_1$  do
10:        I'  $\leftarrow$  I' +  $\lambda \cdot \partial(\sum_{n \in N} \text{Neuron}(n, I')) / \partial I'$ 
11:        O'  $\leftarrow$  Run(DNN, I')
12:        if Cov(O')  $\not\subseteq$  C  $\wedge$  Constraint(I, I') then
13:          W  $\leftarrow$  W  $\cup$  {I'}
14:          C  $\leftarrow$  C  $\cup$  Cov(O')
15:   until testing budget expires (e.g. timeout)
16:   return |C|

```

---

is to find a set of input vectors  $T = \{I_1, \dots, I_n\}$  that collectively maximize the coverage:

$$\text{Find } T = \{I_1, \dots, I_n\} \text{ that maximizes } |\bigcup_{I_i \in T} \text{Cov}(\text{Run}(DNN, I_i))|.$$

Existing white-box testing techniques aim to solve this optimization problem with a search algorithm guided by the gradient of internal neurons [8, 12, 24]. Algorithm 1 presents the general architecture that encapsulates the existing white-box techniques. The algorithm takes a neural network (*DNN*), an initial input vector (*I*), and a coverage metric (Cov). At line 2, the set *C* is initialized to the set of neurons covered by the initial input. Initially, the worklist *W* is a singleton set {*I*} (line 4). The algorithm starts by selecting an input *I'* from the worklist *W* (line 6). At line 8, the function Strategy takes the *DNN* and then selects a set *N* of neurons (in experiments, we select 10 neurons). At line 10, the algorithm calculates the gradient of the selected neurons *N*, and then uses the gradient to adjust the input in the direction of increasing the selected output values; the algorithm generates a new input *I'* by adding the gradient multiplied by the learning rate  $\lambda$ , where Neuron is a function modeled by the neuron  $n_{k,i}$  that takes an input *I* and calculates the output of the neuron  $n_{k,i}$  which can be written as follows:

$$\text{Neuron}(n_{d,i}, I) = \begin{bmatrix} w_{d,1,i} \\ \vdots \\ w_{d,t_{d-1},i} \end{bmatrix}^T (\text{Layer}_{d-1} \circ \dots \circ \text{Layer}_2)(I)$$

At line 12, the algorithm checks whether the new input *I'* is able to cover new neurons, and whether it satisfies a given constraint (Constraint) on test cases. For example, Constraint(*I*, *I'*) is true if L2 distance between *I* and *I'* is less than a certain threshold. If both conditions are met, we add the new input vector *I'* to the worklist *W*, and also add the newly covered neurons to the set *C* (lines 13-14). In the inner loop at lines 9-14, the algorithm repeatedly generates new input vectors *I'* for  $\eta_1$  times with the same gradient, where  $\eta_1$  is typically small (e.g.  $\eta_1 = 3$  in our experiments). The procedure described above is repeated until the testing budget expires. Upon termination, the number of covered neurons (|*C*|) is returned.

The existing white-box techniques differ essentially in the choice of the neuron-selection strategy (Strategy). For example, Deep-Xplore [24] uses a strategy that randomly chooses neurons that have never been activated, and DLFuzz [12] supports four neuron-selection strategies. In this paper, we show that the neuron-selection strategy is a key component of white-box testing and propose a new testing technique that adaptively employs neuron-selection strategies.

### 3 OUR TECHNIQUE

In this section, we present our white-box testing technique for neural networks. The key feature is continuously learning the neuron-selection strategy while performing Algorithm 1. To do so, we present a parameterized neuron-selection strategy and an online learning algorithm.

#### 3.1 Parameterized Neuron-Selection Strategy

We first parameterize the neuron-selection strategy. The parameterized strategy, denoted  $\text{Strategy}_p$ , selects the neurons based on the parameter, *p*, which is a vector of real numbers. We define the strategy as follows:

$$\text{Strategy}_p(DNN) = \operatorname{argmax}_{S \subseteq \bigcup_{d \in \{2, \dots, D-1\}} L_d \wedge |S|=m} \left( \sum_{n \in S} \text{score}_p(n) \right)$$

Intuitively, the strategy takes as input all neurons in the hidden layers (i.e.  $\bigcup_{d \in \{2, \dots, D-1\}} L_d$ ), and selects the top-*m* neurons according to their scores. The number of selected neurons (*m*) is a predetermined hyper-parameter (in experiments, we set *m* to 10).

For scoring, we use a simple method based on linear combination. To score each neuron in *DNN*, we first transform each neuron in the network model *DNN* into a feature vector. A single feature is a boolean function on neurons:

$$F_i : \text{Neurons} \rightarrow \{0, 1\}$$

where *Neurons* denotes the set of all neurons in the model *DNN*. For instance, a feature may check whether neurons have been activated or not. We designed 29 features describing properties of neurons. With the 29 features, we can convert each neuron *n* into a 29-dimensional boolean feature vector as follows:

$$F(n) = \langle F_1(n), F_2(n), \dots, F_{29}(n) \rangle.$$

After the transformation, we can score each neuron by calculating an inner product of the feature vector *F(n)* and the parameter vector *p* that is also 29-dimensional:

$$\text{score}_p(n) = F(n) \cdot p.$$

After calculating the score of each neuron, we select the *m* neurons with the highest scores.

**Neuron Features.** We designed 29 atomic features that describe characteristics of the neurons in a neural network. We focused on designing features with simplicity and generality. These features in Table 1 are categorized into two classes: 17 constant and 12 variable features. The key difference between constant and variable features is whether the boolean value of the feature changes during neural network testing; the value of constant feature for the same neuron does not change while the value of variable feature may change.

**Table 1: Neuron features**

#	Description
1	Neuron located in front 25% layers
2	Neuron located in front 25-50% layers
3	Neuron located in front 50-75% layers
4	Neuron located in front 75-100% layers
5	Neuron in a normalization layer
6	Neuron in a pooling layer
7	Neuron in a convolution layer
8	Neuron in a dense layer
9	Neuron in an activation layer
10	Neuron in a layer with multiple input sources
11	Neuron that does not belong to of 5-10 features
12	Neuron with top 10% weights
13	Neuron with weights between top 10% and 20%
14	Neuron with weights between top 20% and 30%
15	Neuron with weights between top 30% and 40%
16	Neuron with weights between top 40% and 50%
17	Neuron with weights in bottom 50%
18	Neuron activated when an adversarial input is found
19	Neuron never activated
20	Neuron with the number of activations (top 10%)
21	Neuron with activation numbers (top 10-20%)
22	Neuron with activation numbers (top 20-30%)
23	Neuron with activation numbers (top 30-40%)
24	Neuron with activation numbers (top 40-50%)
25	Neuron with activation numbers (top 50-60%)
26	Neuron with activation numbers (top 60-70%)
27	Neuron with activation numbers (top 70-80%)
28	Neuron with activation numbers (top 80-90%)
29	Neuron with activation numbers (top 90-100%)

Constant features consist of 11 features describing the layer where the neuron is located and 6 features describing the neuron itself. Specifically, the constant features 1-4 describe the relative positions of layers in the neural network. We designed the features 5-11 describing the layer type to check the importance of different layer types. Likewise, we designed the 6 constant features to evaluate the importance of the weight each neuron has; we did not deliberately divide the neurons having 50% bottom weights that are likely to be redundant. All constant features can be extracted from the given neural network directly without any calculation.

The variable features 18-29 describe the properties of neurons, where these features continuously change over the testing procedure. We designed feature 18 to check whether there are key neurons in generating adversarial inputs. Depending on the number of activations for each neuron, we have designed features 19-29. All the variable features can be easily extracted with little overhead. We also reflected the key insights of existing white-box testing tools. For example, features 12 and 19 came from the strategies used in DLFuzz [12] and DeepXplore [24], respectively.

### 3.2 White-Box Testing with Online Learning

We now describe our white-box testing technique (Algorithm 2) that adaptively learns and changes neuron-selection strategies based on

**Algorithm 2** Our White-box Testing with Online Learning

---

```

1: procedure ADAPT( $DNN, I, Cov$ )
2:    $C \leftarrow Cov(Run(DNN, I))$ 
3:    $P \leftarrow \{p_1, \dots, p_{\eta_2} \mid p_i \sim \mathcal{U}([-1, 1]^{29})\}$ 
4:    $H \leftarrow \emptyset$ 
5:   repeat
6:     for all  $p \in P$  do
7:        $C_p \leftarrow \emptyset$ 
8:        $W \leftarrow \{I\}$ 
9:       while  $W \neq \emptyset$  do
10:         $I' \leftarrow$  Pick an input from  $W$ 
11:         $W \leftarrow W \setminus \{I'\}$ 
12:         $N \leftarrow Strategy_p(DNN)$ 
13:        for  $t = 1$  to  $\eta_1$  do
14:           $I' \leftarrow I' + \lambda \cdot \partial(\sum_{n \in N} Neuron(n, I')) / \partial I'$ 
15:           $O' \leftarrow Run(DNN, I')$ 
16:          if  $Cov(O') \not\subseteq C \wedge Constraint(I, I')$  then
17:             $W \leftarrow W \cup \{I'\}$ 
18:             $C \leftarrow C \cup Cov(O')$ 
19:             $C_p \leftarrow C_p \cup Cov(O')$ 
20:           $H \leftarrow H \cup \{(p, C_p)\}$ 
21:           $H \leftarrow Pop(H, \eta_3)$ 
22:         $P \leftarrow Learning(H)$ 
23:      until testing budget expires (e.g. timeout)
24:    return  $|C|$ 

```

---

the data accumulated during neural-network testing. In parameterized neuron-selection strategy, a 29-dimensional parameter vector of real-numbers corresponds to a single neuron-selection strategy. Hence, we consider a set of parameter vectors as a set of neuron-selection strategies in Algorithm 2.

**Overall Testing Process.** Unlike Algorithm 1, Algorithm 2 maintains the set  $P$  of neuron selection strategies in the outer loop at lines 6-21, and changes the set  $P$  based on the accumulated data  $H$  at line 22. The input and output of Algorithm 2 are identical to the ones in Algorithm 1. At line 3, the algorithm initially generates  $\eta_2$  random neuron-selection strategies, where  $\mathcal{U}$  denotes the probability density function of the continuous uniform distribution. At lines 8-18, for each strategy  $p$ , Algorithm 2 performs exactly the same as in Algorithm 1; the algorithm selects the set  $N$  of neurons (line 12), and generates the new input  $I'$  by adding the gradient of the selected neurons  $N$  (line 14). Then, algorithm checks whether the new input is able to cover new identifiers and satisfy the distance constraint (line 16). At line 20, unlike Algorithm 1, our technique keeps updating the data  $H$  during neural network testing, where each element in  $H$  denotes a tuple of the strategy  $p$  and the covered identifiers  $C_p$  by the strategy. At line 21, to use the latest information accumulated in  $H$  as learning data, the algorithm maintains the size of the data  $H$  as  $\eta_3$  by applying the function  $Pop$ ; the function returns the set  $H$  with the most recently accumulated  $\eta_3$  elements. Then, it newly generates the set  $P$  with the  $Learning$  function (line 22).

**Learning.** The idea of Learning is to identify “crucial” neuron-selection strategies from the accumulated data  $H$  and then generate

new strategies by combining the features of those crucial strategies. To do so, the learning procedure (Learning) consists of two steps: Extract and Combine.

In the Extract step, the goal is to collect the set  $S$  of crucial strategies from  $H$ , where the size of  $S$  is fixed by a hyper-parameter  $\eta_4$ . Collecting  $S$  is done in the following two steps. First, we collect the set  $S_1$  of strategies from  $H$  that collectively maximize the number of covered identifiers. Let  $H^*$  be all the possible such subsets of  $H$  whose sizes are bounded by  $\eta_4$ :

$$H^* = \operatorname{argmax}_{H' \subseteq H \wedge |H'| \leq \eta_4} \left| \bigcup_{(\_, C_p) \in H'} C_p \right|$$

where  $\operatorname{argmax}$  produces the set of all arguments that maximize the given objective, i.e.,  $\operatorname{argmax}_{g(x)} f(x) = \{x \mid g(x) \wedge \forall y. f(y) \leq f(x)\}$ .<sup>3</sup> With  $H^*$ , we can define the set  $S_1$  of strategies as follows:

$$S_1 = \{p \mid (p, \_) \in \operatorname{argmin}_{H' \in H^*} |H'|\}$$

where  $\operatorname{argmin}$  arbitrarily picks a smallest  $H' \in H^*$  when it is not unique. Intuitively, the set  $S_1$  denotes the minimum set of strategies that collectively maximize the number of covered identifiers. If the size of the set  $S_1$  is less than  $\eta_4$ , we additionally collect the set  $S_2$  of  $\eta_4 - |S_1|$  strategies with the maximal number of covered identifiers. That is, we aim to find the set  $S_2$  defined as follows:

$$S_2 = \{p \mid (p, \_) \in \operatorname{argmax}_{H' \subseteq H \wedge (|H'| = \eta_4 - |S_1|)} \sum_{(\_, C_p) \in H'} |C_p|\}.$$

Note that, because we collect  $S_1$  and  $S_2$  from the set  $H$ , a single strategy  $p$  can be an element of both  $S_1$  and  $S_2$ . In this case, we assume such a strategy is more promising than the strategies exclusively contained in one of the two sets. Thus, we define the final set  $S$  to be the following:

$$S = \{\{p \mid p \in S_1\} \cup \{p \mid p \in S_2\}\}$$

where the notation  $\{\}$  indicates multisets that allow duplicated elements.

For example, suppose that the accumulated data  $H$  is the following:

$$H = \{(p_1, \{i_1, i_2, i_3, i_4\}), (p_2, \{i_2, i_4\}), (p_3, \{i_1, i_3\}), \\ (p_4, \{i_3, i_4, i_5\}), (p_5, \{i_1, i_3, i_4\}), (p_6, \{i_5, i_6\})\}$$

where each element in  $H$  consists of a tuple of a strategy and the corresponding covered identifiers. When the hyper-parameter  $\eta_4$  is 3, the set  $S_1$  of crucial strategies is as follows:

$$S_1 = \{p_1, p_6\}$$

where  $S_1$  is the minimum set of the strategies that collectively maximize the number of covered identifiers (i.e.,  $\{i_1, i_2, i_3, i_4, i_5, i_6\}$ ). Then, we additionally collect the set  $S_2$  with  $\eta_4 - |S_1|$  strategies with the maximal number of covered identifiers, where  $\eta_4$  is 3 and  $|S_1|$  is 2. That is, the set  $S_2$  corresponds to a singleton set  $\{p_1\}$  where  $p_1$  has the maximal coverage in  $H$ . Finally, we can obtain the set  $S$  that includes all the strategies in  $S_1$  and  $S_2$ :

$$S = \{p_1, p_6, p_1\}.$$

The intuition behind this step is to extract the set  $S$  capturing the core knowledge in the accumulated data  $H$ .

<sup>3</sup>Elsewhere in this paper, we assume  $\operatorname{argmax}$  and  $\operatorname{argmin}$  return a single argument since the result is a singleton set or the choice is unimportant.

In the Combine step, we generate new strategies by combining the features of the strategies in  $S$ , where the number of strategies to generate is given as the hyper-parameter  $\eta_2$ . This step has a genetic-algorithm flavor and repeats the next four phases until we have  $\eta_2$  new strategies.

- (1) We randomly sample two strategies from  $S$ . For instance, suppose that the strategies,  $p_1$  and  $p_6$ , are sampled, where we assume each strategy is represented by a 5-dimensional vector of real-numbers as follows:

$$\begin{aligned} p_1 &= \langle 0.2, 0.6, -0.3, 0.9, -0.4 \rangle \\ p_6 &= \langle -0.1, -0.7, 0.2, 0.5, -0.8 \rangle. \end{aligned}$$

- (2) We generate a new strategy  $p'$ , a parameter vector, by mixing the two selected strategies. The  $i$ -th component of the new vector  $p'$  is chosen randomly from the  $i$ -th components of the two vectors. For instance, if the first and third components of  $p'$  are obtained from  $p_1$ , and other components are from  $p_6$ , the newly generated strategy  $p'$  will be as follows:

$$p' = \langle 0.2, -0.7, -0.3, 0.5, -0.8 \rangle.$$

- (3) We add a small random noise to each component of the newly generated strategy  $p'$ :

$$p' = \langle 0.25, -0.72, -0.3, 0.51, -0.82 \rangle$$

where the noise is sampled from the normal distribution  $\mathcal{N}(0, 0.2^2)$ . This step aims to enable exploration of the space of possible strategies without changing the new strategy too much; hence, we use a relatively small standard deviation.

- (4) Finally, we clip the newly generated strategy  $p'$  to ensure that the new one is in the proper range (e.g.,  $[-1, 1]$ <sup>29</sup>).

As the entire procedure (Algorithm 2) is going on, our technique is able to generate effective neuron-selection strategies based on the accumulated data  $H$ , thereby achieving high coverage.

**Hyperparameters.** Our algorithm involves five hyperparameters  $m$ ,  $\eta_1$ ,  $\eta_2$ ,  $\eta_3$ , and  $\eta_4$ . The first hyperparameter  $m$  denotes the number of neurons that the strategy (Strategy) selects for the gradient calculation. Assigning the appropriate value to  $m$  is important. For instance, with small  $m$  (e.g.  $m = 3$ ), we failed to make any notable influence on the inputs; with large  $m$  (e.g.  $m = 30$ ), the generated input was often too far from the original one. In the experiments, with trial and error, we set  $m$  to 10 (we used the same value for other white-box testing techniques: DLFuzz and DeepXplore). The second hyper-parameter  $\eta_1$  is the number of times that the set of selected neurons is used for generating new inputs. Note that because Algorithm 2 continuously adds the gradient value  $\eta_1$  times to new input  $I'$  on line 14, the larger the value of  $\eta_1$ , the greater the distance between the original input  $I$  and the new one  $I'$ . We manually set the  $\eta_1$  to 3 so that the distance is not too far. The third one  $\eta_2$  represents the number of new strategies to generate after learning at line 22 in Algorithm 2. Intuitively, as the size of the  $\eta_2$  increases, the number of learning trials (line 22) during the entire testing budget decreases. That is,  $\eta_2$  determines how often the algorithm performs the learning procedure. In experiments, we set  $\eta_2$  to 100 to perform the learning procedure multiple times during the total testing budget. Forth,  $\eta_3$  denotes the size of the set  $H$  to be used as the learning data (line 21 in Algorithm 2). For instance, if  $\eta_3$  is 10, we only use the most recently accumulated 10

**Table 2: Datasets and DNN models**

Dataset	Model	# of Neurons	# of Layers	Accuracy
MNIST	LeNet-4	148	9	0.985
	LeNet-5	268	10	0.988
ImageNet	VGG-19	16,168	26	0.713
	ResNet-50	94,123	177	0.764

elements in  $H$  as learning data. We set  $\eta_3$  to 300 in our experiments. Fifth,  $\eta_4$  denotes the number of effective strategies to select from the learning data  $H$  for generating new strategies. Intuitively, if the size of the  $\eta_4$  is very small (e.g.,  $\eta_4=2$ ), we will generate new strategies that are similar to the most effective top-2 ones in the learning data. In experiments, we set  $\eta_4$  to 50. In this work, we manually tuned these hyper-parameters and left automatic tuning as future work.

## 4 EXPERIMENTS

We implemented our technique in a tool, called **ADAPT**, using Python 3.6.3, Tensorflow 1.14.0 [1], and Keras 2.2.4 [6] without any modification of the frameworks. We evaluated **ADAPT** to answer the following research questions:

- **Coverage:** How effectively does **ADAPT** increase coverage metrics across various DNN models and datasets? How does it compare to existing testing techniques?
- **Adversarial Inputs:** How effectively does **ADAPT** find adversarial inputs compared to existing techniques? Is there strong correlation between coverage metrics and defects?
- **Learned Insight:** Is there any learned insight that should be considered while testing deep neural networks?

All experiments were done on a machine with two Intel Xeon Processors (E5-2630), 192GB RAM, and a NVIDIA GTX 1080 GPU.

### 4.1 Experimental Setup

We used two datasets and four neural network models in Table 2, which have been used in prior work [8, 12, 24, 34]. MNIST [17] is a classical dataset of hand-written digits with 10 classes and ImageNet [7] is a huge collection of real-world images with 1,000 class labels. For each dataset, we used two pre-trained models: LeNet-4 and LeNet-5 [17] for MNIST, and VGG-19 [29] and ResNet-50 [13] for ImageNet. In particular, VGG-19 and ResNet-50 are widely used in practice when, for example, extracting the embedding of the images in various tasks such as style transfer [10, 14], image captioning [35], and visual question answering [9].

We compared **ADAPT** with five state-of-the-art testing techniques for DNNs: four white-box and one grey-box approaches. The white-box approaches include DeepXplore [24], two instances of DL-Fuzz [12], and a random baseline. These techniques mainly differ in the neuron-selection strategy (Strategy) in Algorithm 1. DeepXplore randomly selects unactivated neurons. DeepXplore originally performs differential testing requiring multiple DNNs without labeled data but we used it with a single DNN and labeled data in our experiments. We instantiated DL-Fuzz with two strategies: DL-Fuzz<sub>Best</sub> denotes the strategy that performed best in the prior

work [12] and DL-Fuzz<sub>RR</sub> the strategy that combines the three proposed strategies in a round-robin fashion, excluding one strategy that is not compatible with Top- $k$  Neuron Coverage (TKNC) [18]. We included a random strategy Random, which selects neurons randomly. All white-box testing tools select 10 neurons for gradient calculation, as we mentioned earlier. We also compare TensorFuzz [22], an available state-of-the art grey-box testing tool.

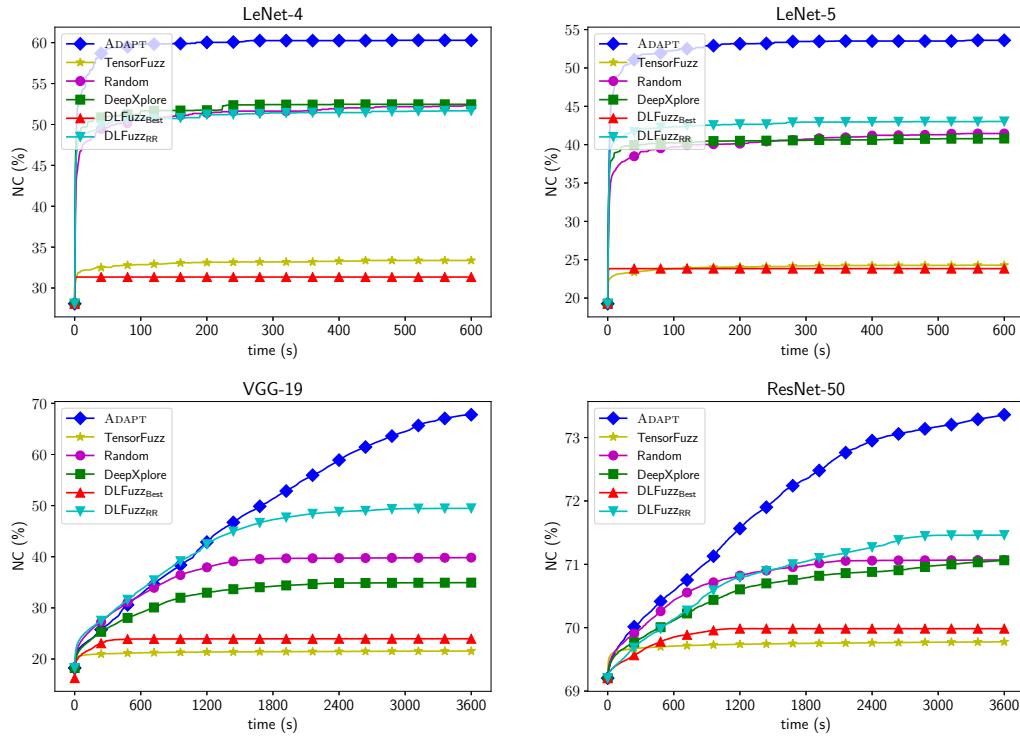
We used two well-known coverage metrics: Neuron Coverage (NC) [24] and Top- $k$  Neuron Coverage (TKNC) [18]. For the former, we set  $\theta$  to 0.5 and 0.25 for small (LeNet-4, LeNet-5) and large (VGG-19, ResNet-50) models, respectively. For the latter, we set  $k$  to 3 and 30 for small and large models, respectively. As an initial input, we used 20 inputs randomly selected from the corresponding testset of datasets for two small models, LeNet-4 and LeNet-5, and 10 randomly chosen images from the 2002 test images provided by DeepXplore [24] for two large models, VGG-19 and ResNet-50. All images are initially correctly classified, which means two models for each dataset classify the images to the same labels. For each input, we allocated 10 minutes and 1 hour as a testing budget for small and large models, respectively. Note that we allocated the same testing budget for **ADAPT** (Algorithm 2) and other existing tools (Algorithm 1); that is, the learning cost in **ADAPT** is included in the budget. All the techniques, including **ADAPT**, performed testing while maintaining the L2-distance between the initial and mutated inputs within 0.05 on average. All numerical values are average over all images tested, except the values with explicit mentions.

### 4.2 Coverage

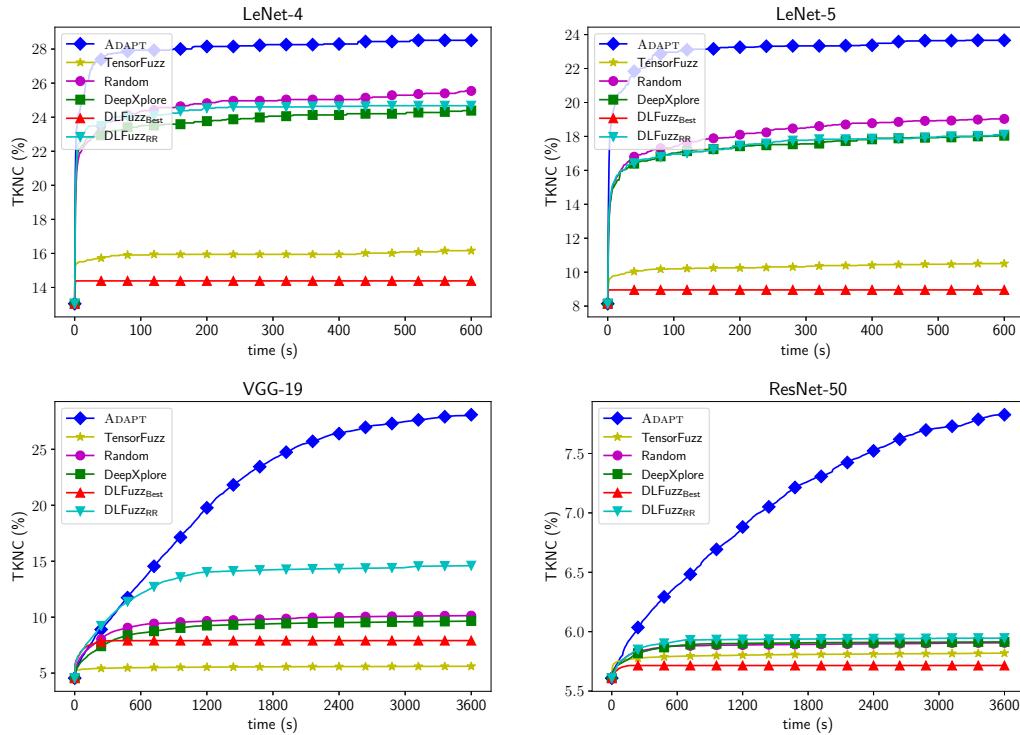
Figure 1 shows that **ADAPT** can achieve remarkably higher coverage than existing techniques across *all metrics and DNN models*. For example, **ADAPT** was able to achieve 67.8% NC on VGG-19 while the second best technique (DL-Fuzz<sub>RR</sub>) achieved 49.5%; **ADAPT** covered almost 3,000 more neurons than DL-Fuzz<sub>RR</sub> during the same time period. Likewise, **ADAPT** managed to achieve 7.8% TKNC for ResNet-50, while no other existing tools can achieve the higher coverage than 6%, even though the initial coverage is 5.6%. Our tool successfully achieved the highest coverage on small models as well. For instance, ours increased NC by 7.8% and 10.6% more, compared to the second best technique of each model: DeepXplore (Lenet-4) and DL-Fuzz<sub>RR</sub> (Lenet-5).

Compared to the larger models (VGG-19 and ResNet-50), the smaller models (LeNet-4 and LeNet-5) converged remarkably fast. This is because a small model requires much less time for its execution than the larger model. For instance, we observed that our tool generated about 3 times more inputs when testing LeNet-4 with NC than when testing ResNet-50 with NC, even though we allocated 6 times more time budget to larger models than smaller ones.

Note that, except for **ADAPT**, existing testing tools have unstable performance. For example, DL-Fuzz<sub>RR</sub> achieved the highest neuron coverage for LeNet-5, but DeepXplore outperforms other existing techniques for LeNet-4. Regarding TKNC, all existing techniques are inferior even to the random baseline (Random) for LeNet-4 and LeNet-5. Figure 1 also shows that TensorFuzz, a grey-box technique, is overall less effective than white-box techniques.



(a) Average neuron coverage (NC) achieved by each technique on four models and two datasets

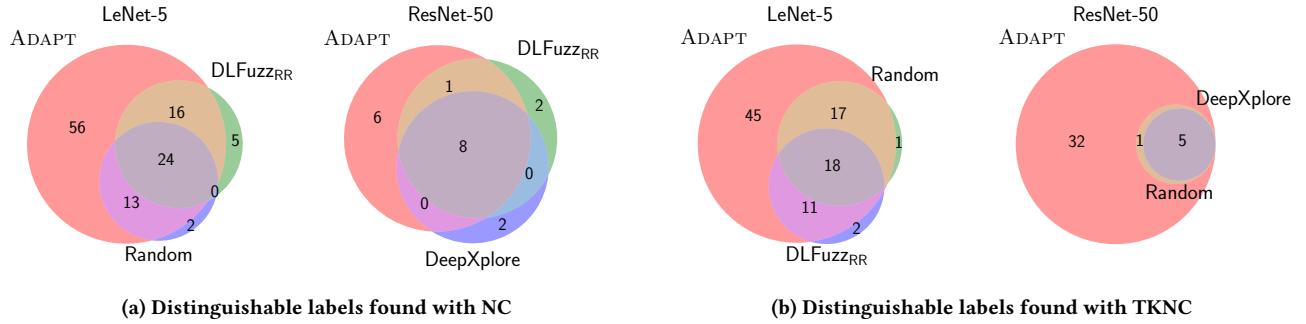


(b) Average Top-k neuron coverage (TKNC) achieved by each technique on four models and two datasets

Figure 1: Effectiveness for increasing NC and TKNC metrics

**Table 3: Effectiveness for finding adversarial inputs**

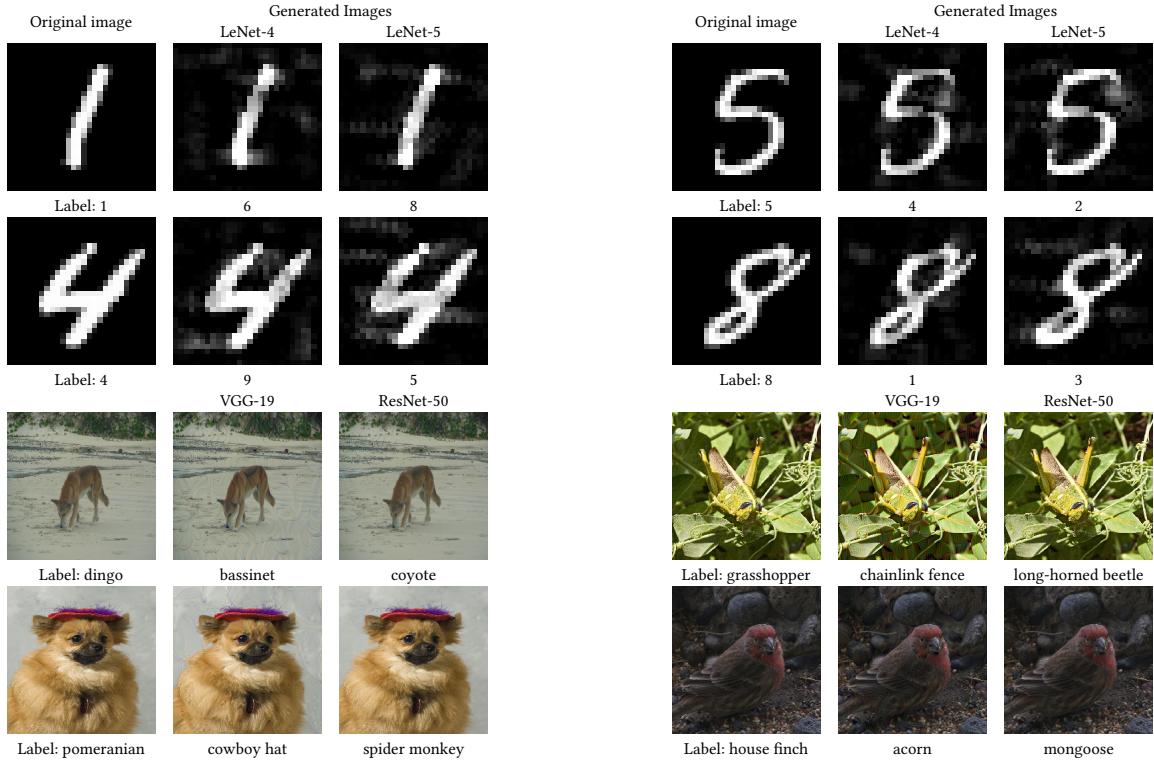
Dataset	Model	Metric	With Neuron Coverage				With Top- $k$ Neuron Coverage				
			Technique	Mutations	Adv. Inputs	Labels	Seeds	Mutations	Adv. Inputs	Labels	Seeds
MNIST	LeNet-4	ADAPT	31888.9	<b>950.9</b>	73	<b>20/20</b>		38320.4	955.8	<b>48</b>	<b>16/20</b>
		TensorFuzz	74881.0	0.0	0	0/20		73515.8	0.0	0	0/20
		Random	34988.6	192.3	38	17/20		32394.8	178.8	33	14/20
		DeepXplore	38191.6	30.2	26	14/20		38083.0	11.1	15	7/20
		DLFuzzBest	33895.5	542.2	3	3/20		32857.7	<b>1079.7</b>	1	1/20
		DLFuzzRR	34223.8	423.9	33	16/20		32934.0	41.1	20	12/20
ImageNet	LeNet-5	ADAPT	35601.6	<b>3974.5</b>	<b>109</b>	<b>20/20</b>		36483.6	<b>3202.9</b>	<b>91</b>	<b>20/20</b>
		TensorFuzz	89290.4	0.0	0	0/20		90024.1	0.0	0	0/20
		Random	30686.7	225.3	39	17/20		32994.7	215.8	36	16/20
		DeepXplore	36188.1	39.0	37	17/20		34952.9	88.9	23	12/20
		DLFuzzBest	31659.3	2.0	4	4/20		33006.6	0.2	2	2/20
		DLFuzzRR	32742.3	183.8	45	18/20		33776.3	182.6	31	15/20
ImageNet	VGG-19	ADAPT	12028.6	<b>5301.0</b>	<b>265</b>	<b>10/10</b>		11926.1	3322.7	<b>300</b>	<b>10/10</b>
		TensorFuzz	15173.1	0.0	0	0/10		12911.8	0.0	0	0/10
		Random	12778.1	1135.0	56	6/10		13005.6	126.2	10	4/10
		DeepXplore	10210.6	695.4	25	5/10		9077.9	163.6	15	4/10
		DLFuzzBest	12888.5	4713.2	9	8/10		12246.3	<b>5091.5</b>	6	5/10
		DLFuzzRR	11844.7	2131.0	100	9/10		12139.4	673.2	65	7/10
ImageNet	ResNet-50	ADAPT	8469.3	<b>1925.1</b>	<b>15</b>	5/10		8302.2	<b>1721.2</b>	<b>38</b>	<b>6/10</b>
		TensorFuzz	9237.0	0.0	0	0/10		9464.0	0.0	0	0/10
		Random	9256.5	713.0	10	5/10		8916.4	137.1	5	3/10
		DeepXplore	6541.3	658.7	10	5/10		6832.9	146.3	6	4/10
		DLFuzzBest	8357.8	597.6	8	5/10		9087.0	61.1	3	3/10
		DLFuzzRR	8435.0	1095.2	11	<b>6/10</b>		9026.4	167.5	3	3/10

**Figure 2: The number of distinguishable labels found**

### 4.3 Adversarial Inputs

While conducting the experiments in terms of coverage in Section 4.2, we found that ADAPT is highly effective in finding adversarial inputs as well. Table 3 reports the average number of mutations per image (Mutations), the average number of adversarial inputs found per image (Adv. Inputs), the total number of incorrectly classified labels (Labels), and the number of initial inputs (Seeds) from which adversarial inputs were found. An incorrectly classified label consists of an original image and a found label. That is, we consider two adversarial inputs differently, which classified

into the same label but came from different input source. Overall, ADAPT outperforms existing techniques in almost all metrics; on average, ADAPT succeeded to find 4.9 times more adversarial inputs, 6.4 times more incorrectly classified labels, and 2.0 times more seed images than existing techniques. Out of 24 metrics, ADAPT ranked at the first place in 21 metrics. In terms of the number of the labels found, no existing techniques could beat ADAPT. ADAPT is also able to find adversarial inputs from 86% of the given inputs on average, while the best technique (DLFuzzRR) among existing techniques can found adversarial inputs from only 69%. DLFuzzBest



**Figure 3: Images with incorrectly classified labels found exclusively by ADAPT.**

oddly found a lots of adversarial inputs, while the number of labels and the number of seed inputs from which adversarial inputs are found are relatively small. This is because DLFuzzBest selects most activated neurons, which are fixed as the testing procedure goes by, which results in generating a huge amount of similar inputs without any variety. Although the grey-box testing tool, TensorFuzz, which does not calculate the gradients of the outputs of internal neurons, is on average 1.7 times faster than ADAPT, it failed to generate any adversarial inputs across all experiments.

Figure 2 shows that ADAPT successfully found the various incorrect labels compared to existing techniques. The Venn diagram represents the relationship between first (red), second (green), and third (blue) ranked tools in terms of variety of labels. With NC, ADAPT could find 69 more incorrect labels while testing LeNet-5 compared to DLFuzzRR, which is second best technique among existing techniques. For ResNet-50, ADAPT found all the labels that second best (DeepXplore) and third best (Random) found and 32 labels more. In total, ADAPT found 611 more labels that no other techniques could find. Figure 3 shows the some examples with incorrectly classified labels found exclusively by ADAPT (all existing testing techniques totally failed to find those incorrect labels). In particular, the adversarial images found for VGG-19 and ResNet-50 were visually indistinguishable from the original ones.

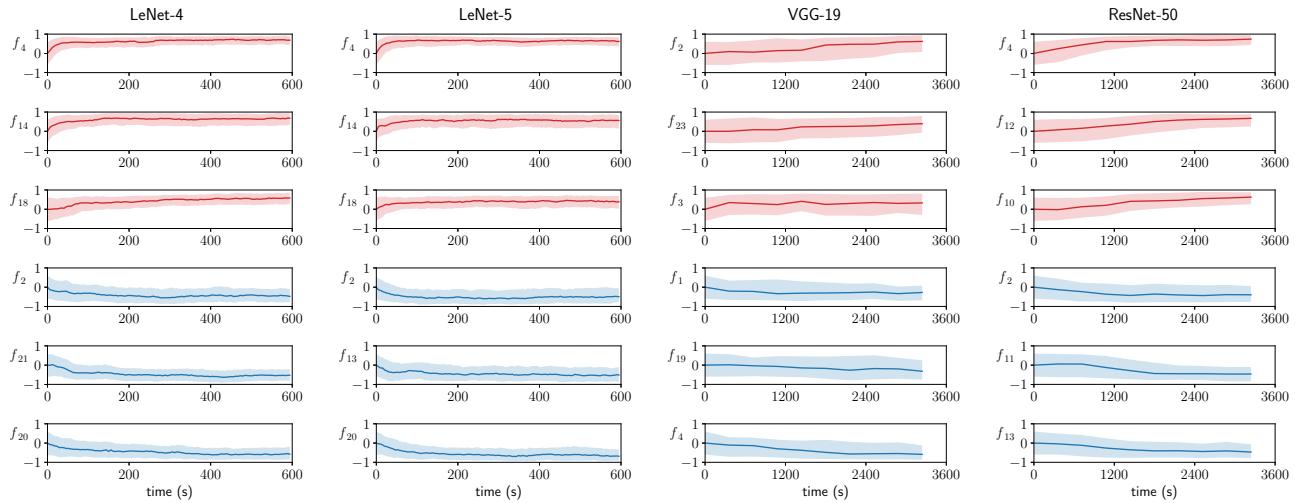
**Correlation between Coverage and Adversarial Inputs.** From Figure 1 and Table 3, we can notice that coverage and variety of

labels are highly correlated, while coverage and the number of adversarial inputs are not correlated strongly. In testing VGG-19 with NC, the order of coverage value of each technique exactly matches the order of the number of the labels found by each technique.

In Figure 2, we can find that the effectiveness of coverage metrics is correlated with the size of the model to test. NC is more helpful than TKNC in finding diverse inputs for small models, like LeNet-4 and LeNet-5, TKNC is more useful in finding diverse inputs for larger models, like VGG-19 and ResNet-50. For instance, ADAPT could find 109 incorrect labels from LeNet-5 with NC, but only 91 labels with TKNC. On the other hand, ADAPT only could find 15 labels from ResNet-50 with NC, but found 38 labels with TKNC.

#### 4.4 Learned Insights

Figure 4 shows the changes of the weights of the top-3 (red) and bottom-3 (blue) features of the strategies that our algorithm collected while testing each model with NC, where  $f_i$  indicates the  $i$ -th element of the parameterized neuron-selection strategies. The top features mean the properties of neurons which should be selected to increase coverage, and the bottom features are characteristics that neurons that should not be selected have. The solid lines represent the medium value of the feature and the colored areas represent the value from the top 20% to 80% of the collected strategies. As testing goes by, the solid lines were being skewed toward one direction, and the colored areas were becoming narrower, which means that



**Figure 4: Top-3 and bottom-3 features learned by our algorithm during testing each model with NC**

the learning algorithm extracts the characteristics of strategies, which increase coverage, well.

$f_4$ , which indicates the layers located in the back 25% of the network, is learned as the most important feature that neurons to select should have, and  $f_2$ , which indicates the layers located relatively front of the network, is included in bottom-3 features in most cases (LeNet-4, LeNet-5, and ResNet-50). This trend indicates that neurons with higher expressive ability, which means that neurons are located in deeper convolutional layers [38], should be selected while testing the deep neural networks. This can explain the results from VGG-19, which has a few fully-connected layers at the back of the network; neurons with  $f_2$  and  $f_3$ , which represent the convolutional layers that located at relatively back, should be considered, and neurons with  $f_4$ , which includes the last fully-connected layers, should be discarded.

$f_{10}$ , which indicates the layers with multiple input sources and no other networks have, is one of the top-3 features that should be selected for testing ResNet-50, while no other features that related to the type of the layer did not show any trend across all experiments. This implies that the layers with multiple input sources should be considered when testing a deep neural network that contains them as their components; the neurons that located in the layers with residual connection in ResNet-50 are examples of the neuron with multiple input sources.

LeNet-4 and LeNet-5 showed very similar results; their top-3 features and two of bottom-3 features exactly match. From these results, we can infer that neurons with moderately high weights ( $f_{14}$ ) and neurons that activated when the given objective is satisfied ( $f_{18}$ ) should be considered carefully while testing small models, like LeNet-4 and LeNet-5. In addition, neurons that activated a lot ( $f_{20}$  and  $f_{21}$ ) should not be selected. These are not consistent with DLFuzz's selection strategies [12] which selects the neurons that activated a lot or neurons with biggest weights; on the other hand, the other strategy of DLFuzz ( $f_{12}$ , neurons with top 10% weights) is included in top-3 strategies in the results of ResNet-50.

The overall trends of the features of the collected strategies during test procedure with TKNC are similar with those with NC. For instance, neurons with higher expressiveness are important in both cases. This represents that there are some common characteristics that two metrics share.

## 5 RELATED WORK

**DNN Testing.** Unlike existing gradient-based white-box techniques, ADAPT uses an adaptive neuron-selection strategy. DeepXplore [24] proposed the first white-box differential testing algorithm to generate inputs which can cause inconsistencies between the set of DNNs. The tool uses gradient ascent as an input generation algorithm, which uses random selection as a neuron selection strategy. The following approach, DLFuzz [12], enabled testing with a single DNN. They use gradient ascent like the former, using four fixed heuristics to select neurons. DeepFault [8] presented a new fault localization-based testing approach by using a neuron-selection strategy based on suspiciousness metric.

Another white-box approach, DeepConcolic [31], tests DNN using concolic testing, which has proven to be effective in small neural networks. However, its applicability to real-world sized networks needs to be examined. DeepMutation [19] analyzes the robustness of the model by mutating the dataset and the model with its self-designed heuristics. It chooses the layers for the heuristics while ADAPT chooses neurons for the gradient calculation. They also have not been confirmed as applicable to real scale models such as ImageNet models. MODE [20] focuses on a different problem; it aims to pick the mis-trained layer and create the input that generates the bug during the learning process.

Grey-box testing techniques are largely based on coverage-guided fuzzing. DeepTest [33] presented a testing method for detecting erroneous behaviors of autonomous car models. They mimic what would happen in the physical world and generate input by applying a set of natural image transformations randomly. DeepHunter [34] performed misbehavior detection of DNNs as well as model quality

evaluation and defect detection in quantization settings based on multiple pluggable coverage criteria feedback. TensorFuzz [22] debugged neural networks by using logit-based coverage and adding additive random noise.

Existing coverage metrics are largely divided into the coverage applied to the weights of the trained model and the coverage applied based on the changes of the weights during the training. In the former case, in addition to NC [24] and TKNC [18], there is a Top- $k$  Neuron Pattern (TKNP) [18] that measures the different kinds of activated scenarios by recording the patterns of top- $k$  neurons per layer. There is also a coverage metric inspired from the classical MC/DC [30]. The latter case includes  $k$ -multisection Neuron Coverage (KMNC) [18], Neuron Boundary Coverage (NBC) [18], Strong Neuron Activation Coverage (SNAC) [18], and Surprise Coverage (SC) [15]. In this paper, we used only the former cases using the pre-trained weights of real-world models. However, our approach is in principle applicable to other coverage metrics as well.

**Using Gradients to Attack DNNs.** Gradients, which can also be used to increase the probability of a particular class, have been used for generating inputs that fool neural networks by generating adversarial examples [3, 11, 16, 23]. They are similar to white-box testing in that they calculate a gradient through an objective function and attempt to find a malfunctioning input. However, they differ from the testing approaches as they aim to find common misbehaving inputs and do not take into account the logic inside the model. The emphasis of testing techniques is on systematically examining the logic of the model and explore various internal states.

**Software Testing with Learning.** Combining software testing and learning is not new. For example, Cha et al. [4, 5] used learning to generate search heuristics of concolic testing automatically. To our knowledge, however, existing works target traditional software and ADAPT is the first tool that uses learning for DNN testing.

## 6 CONCLUSION

Since deep neural networks are used in safety-critical applications, testing safety properties of deep neural networks is important. Although many testing techniques have been introduced recently, there is no technique that is sufficiently effective across different models and coverage metrics. In this paper, we present a new white-box technique, called ADAPT, that performs well regardless of models and metrics, via parameterizing the neuron-selection strategy and learning appropriate parameters online. Experimentally, we demonstrated that ADAPT is significantly more effective than existing white-box and grey-box techniques in increasing coverage and finding adversarial inputs. As future work, we plan to apply our technique to various models, domains, and coverage metrics.

## ACKNOWLEDGMENTS

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair) and Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

## REFERENCES

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, 265–283.
- [2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).
- [3] N. Carlini and D. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy (S&P'17)*, 39–57.
- [4] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 1244–1254.
- [5] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, 235–245.
- [6] François Fleuret et al. 2015. Keras. <https://keras.io>.
- [7] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255.
- [8] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering (FASE'19)*, 171–191.
- [9] Akira Fukui, Dong Huk Park, Daylen Yang, Anna Rohrbach, Trevor Darrell, and Marcus Rohrbach. 2016. Multimodal Compact Bilinear Pooling for Visual Question Answering and Visual Grounding. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP'16)*, 457–468.
- [10] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2016. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR'16)*, 2414–2423.
- [11] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations (ICLR'15)*.
- [12] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jianguang Sun. 2018. DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*, 739–743.
- [13] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, 770–778.
- [14] Xun Huang and Serge Belongie. 2017. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'17)*, 1501–1510.
- [15] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, 1039–1049.
- [16] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *The International Conference on Learning Representations (ICLR'17)*.
- [17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998), 2278–2324.
- [18] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, 120–131.
- [19] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei Xu, Chao Xie, Li Li, Jianjun Zhao, Yang Liu, and Yadong Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. *CoRR* abs/1805.05206 (2018).
- [20] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection.
- [21] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*, 807–814.
- [22] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 4901–4911.
- [23] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *2016 IEEE European Symposium on Security and Privacy*, 372–387.

- [24] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. 1–18.
- [25] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. 2016. Variational Autoencoder for Deep Learning of Images, Labels and Captions. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 2360–2368.
- [26] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. 2017. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225* (2017).
- [27] J. Sekhon and C. Fleming. 2019. Towards Improved Testing For Deep Learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE'19)*. 85–88.
- [28] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550 (2017), 354.
- [29] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations (ICLR'15)*.
- [30] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *CoRR* abs/1803.04792 (2018).
- [31] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. DeepConcolic: Testing and Debugging Deep Neural Networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 111–114.
- [32] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Structural Test Coverage Criteria for Deep Neural Networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE'19)*. 320–321.
- [33] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 303–314.
- [34] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. 146–157.
- [35] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning (ICML '15)*. 2048–2057.
- [36] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo. 2016. Image Captioning with Semantic Attention. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 4651–4659.
- [37] Michał Zalewski. 2007. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>
- [38] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *European conference on computer vision (ECCV'14)*. 818–833.

# DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks

Yang Feng

State Key Lab for Novel Software  
Technology, Nanjing University  
Nanjing, China  
fengyang@nju.edu.cn

Jun Wan

Ant Financial Services Group  
Hangzhou, China  
wukun.wj@antfin.com

Qingkai Shi

The Hong Kong University of Science  
and Technology  
Hong Kong, China  
qshiaa@cse.ust.hk

Chunrong Fang

State Key Lab for Novel Software  
Technology, Nanjing University  
Nanjing, China  
fangchunrong@nju.edu.cn

Xinyu Gao

State Key Lab for Novel Software  
Technology, Nanjing University  
Nanjing, China  
mf1932046@smail.nju.edu.cn

Zhenyu Chen

State Key Lab for Novel Software  
Technology, Nanjing University  
Nanjing, China  
zychen@nju.edu.cn

## ABSTRACT

Deep neural networks (DNN) have been deployed in many software systems to assist in various classification tasks. In company with the fantastic effectiveness in classification, DNNs could also exhibit incorrect behaviors and result in accidents and losses. Therefore, testing techniques that can detect incorrect DNN behaviors and improve DNN quality are extremely necessary and critical. However, the testing oracle, which defines the correct output for a given input, is often not available in the automated testing. To obtain the oracle information, the testing tasks of DNN-based systems usually require expensive human efforts to label the testing data, which significantly slows down the process of quality assurance.

To mitigate this problem, we propose DeepGini, a test prioritization technique designed based on a statistical perspective of DNN. Such a statistical perspective allows us to reduce the problem of measuring misclassification probability to the problem of measuring set impurity, which allows us to quickly identify possibly-misclassified tests. To evaluate, we conduct an extensive empirical study on popular datasets and prevalent DNN models. The experimental results demonstrate that DeepGini outperforms existing coverage-based techniques in prioritizing tests regarding both effectiveness and efficiency. Meanwhile, we observe that the tests prioritized at the front by DeepGini are more effective in improving the DNN quality in comparison with the coverage-based techniques.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397357>

## KEYWORDS

Deep Learning, Test Case Prioritization, Deep Learning Testing.

### ACM Reference Format:

Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: Prioritizing Massive Tests to Enhance the Robustness of Deep Neural Networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397357>

## 1 INTRODUCTION

We are entering the era of deep learning, which has been widely adopted in many areas. Famous applications of deep learning include image classification [13], autonomous driving [2], speech recognition [45], playing games [35], and so on. Although for the well-defined tasks, such as in the case of Go [35], deep learning has achieved or even surpassed the human-level capability, it still has many issues on reliability and quality. These issues could cause significant losses such as in the accidents caused by the self-driving car of Google and Tesla [7, 36, 50].

Testing is considered to be the common practice for software quality assurance. However, testing on DNN-based software is significantly different from conventional software because, while conventional software depends on programmers to manually build up the business logic, DNNs are constructed based on a data-driven programming paradigm. Thus, sufficient test data, with oracle information, is critical for detecting misbehaviors of DNN-based software. Unfortunately, like the testing techniques for conventional software, DNN testing also faces the problem that automated testing oracle is often unavailable. For example, it costs more than 49,000 workers from 167 countries for about 9 years to label the data in ImageNet [8], one of the largest visual recognition datasets containing millions of images in more than 20,000 categories.

Specially, in the context of testing DNN-based systems, software testers often focus on the tests that can cause the system to behave incorrectly, because diagnosing these tests can provide insights into various problems in the program. This fact naturally motivates us to propose a technique to prioritize tests so that fault-inducing tests can be labeled and analyzed before the other tests. In this

manner, we can obtain maximum benefit from human efforts, even the labeling process is prematurely halted at some arbitrary point due to resource limit.

In the past decades, many test prioritization techniques have been proposed for conventional software systems [10, 31, 46]. In these techniques, code coverage is employed as the metric to guide the prioritization procedure. Two main coverage-based techniques are known as coverage-total and coverage-additional test prioritization [46]. A coverage-total method prioritizes tests based on their individual total coverage rate. That is, we prefer a test to the other one if it covers more program elements. A coverage-additional method differs from the coverage-total method in that, it prefers a test if it can cover more program elements that have not been covered by previous tests.

Unfortunately, for DNN-based systems, although several neuron-coverage criteria have been proposed by software engineering researchers [22, 25], the aforementioned coverage-based methods are not effective as expected, due to some new challenges:

- First, these criteria are proposed to measure testing adequacy. It is often not clear how to improve the DNN quality after testing a DNN with these coverage criteria.
- Second, some coverage criteria cannot distinguish the fault detection capability of different tests. Thus, we cannot prioritize them effectively using the coverage-total prioritization method. For example, given a DNN, every test of the DNN has the same top- $k$  neuron coverage rate [22]. As a result, the coverage-total method becomes meaningless using this coverage criterion.
- Third, for most of existing neuron coverage criteria, only a few tests in a test set can achieve the maximum coverage rate of the set. For example, using the top- $k$  neuron coverage [22], we only need about 1% tests in a test set to achieve the maximum coverage rate of the test set. In this case, the coverage-additional method becomes useless because it stops working after prioritizing the first 1% tests.
- Fourth, coverage-additional method usually takes very high time complexity  $O(mn^2)$ , where  $m$  is the number of elements, e.g., neurons, to cover and  $n$  is the number of tests. Since  $n$  and  $m$  are usually very large for a DNN, this method is not scalable.

To overcome the aforementioned problems and effectively improve the DNN quality, in this paper, we propose a test prioritization technique called DeepGini, especially for image-classification DNNs. DeepGini does not prioritize tests as conventional coverage-based approaches but is based on a statistical perspective of DNN. Such a statistical perspective allows us to reduce the problem of measuring misclassification probability to the problem of measuring set impurity [26]. Intuitively, a test is likely to be misclassified by a DNN if the DNN outputs similar probabilities for each class. Thus, this metric yields the maximum value when DNN outputs the same probability for each class. For example, if a DNN outputs a vector  $\langle 0.5, 0.5 \rangle$ , it means that the DNN is not confident about its classification because the test has the same probability (i.e., 0.5) to be classified into the two classes. In this case, the DNN is more likely to make mistakes. In contrast, if the DNN outputs  $\langle 0.9, 0.1 \rangle$ , it implies that the DNN is confident that the test should be classified

into the first class. Compared to the coverage-based approaches, DeepGini has the following advantages:

- Tests are more distinguishable using our metric than existing neuron coverage criteria. This is because it is not likely that different tests have the same output vector but tests usually have the same coverage rate as discussed above.
- It is not necessary for us to record a great deal of intermediate information to compute coverage rate. We prioritize tests only based on the output vector of a DNN. Since it is not necessary for us to understand the internal structure of a DNN, our approach is much easier to use. Meanwhile, it is also more secure because we do not need to look into a DNN and, thus, sensitive information in a DNN is protected.
- The time complexity of DeepGini is the same with the coverage-total approach but much less than the coverage-additional approach. Thus, our approach is as scalable as coverage-total approaches but much more scalable than coverage-additional approaches.
- Tests prioritized at the front by DeepGini are more effective to improve the DNN quality than those prioritized at the back and those prioritized at the front but by coverage-based prioritization techniques.

We notice that our approach requires to run all tests to obtain the output vectors so that the likelihood of misclassification can be calculated. However, we argue that this is not a significant weakness. First, this issue is shared with all coverage-based prioritization methods as they also need to run tests to obtain the coverage rates. Second, the time cost to run a DNN is not time-consuming like training the DNN. Compared to the expensive cost of manually labeling all tests in a messy order, the time cost is completely negligible.

We compare DeepGini with coverage-based methods using existing neuron coverage criteria. The effectiveness of our approach is evaluated from two aspects. First, we compute the value of Average Percentage of Fault-Detection (APFD) [46], which is a standard method of evaluating prioritization techniques. Second, we evaluate if our technique can improve the quality of DNN. To this end, we add the tests prioritized at the front to the training set and compare the accuracy of the re-trained DNN to the original one. The experimental results demonstrate that DeepGini is close to the optimal solution in terms of the value of APFD, and DeepGini is also more effective to improve the DNN quality. In summary, we make the following contributions in this paper:

- We propose an effective and efficient approach, DeepGini, to prioritizing DNN tests.
- We demonstrate that tests prioritized at the front by DeepGini are effective to improve the DNN quality.
- We show the weaknesses of neuron coverage criteria in test prioritization and DNN enhancement.

The remainder of the paper is organized as follows. Section 2 introduces the background knowledge of deep learning and test prioritization. Section 3 presents our approach to prioritizing tests and its application to improving the DNN quality. Section 4 takes us to the empirical study, in which we introduce the settings of the evaluation. Section 5 discusses the experimental results, which demonstrate the effectiveness and efficiency of our approach. Section 6 surveys the related work and Section 7 concludes this paper.

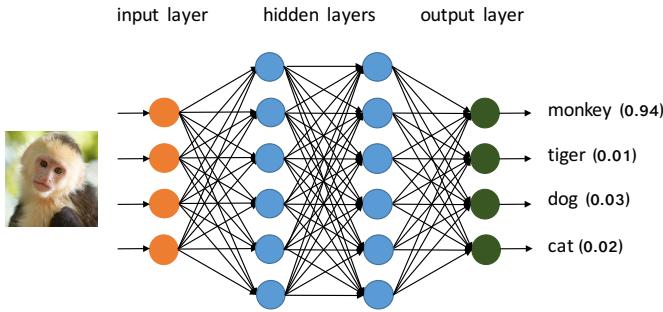


Figure 1: An example to illustrate the DNN structure.

## 2 BACKGROUND

In this section, we introduce the basic knowledge of DNN and conventional test prioritization techniques.

### 2.1 Deep Neural Networks

Classification deep neural networks (DNN) are the core of many deep learning systems. As shown in Figure 1, a DNN consists of multiple layers, i.e., an input layer, an output layer, and one or more hidden layers. Each layer is made up of a series of neurons. The neurons from different layers are interconnected by weighted edges. Each neuron is a computing unit that applies an activation function on its inputs and the weights of the incoming edges. The computed result is passed to the next layer through the edges. The weights of the edges are not specified directly by software developers, but automatically learned by a training process with a large set of labeled training data. After training, a DNN then can be used to automatically classify an input object, e.g., an image with an animal, into its corresponding class, e.g., the animal species.

Suppose we have a DNN that classifies objects into  $N$  classes. Given an input, the DNN will output a vector of  $N$  values, e.g.,  $\langle v_1, v_2, \dots, v_N \rangle$ , each of which represents how much the system thinks the input corresponds to each class. Using a softmax function [9], it is easy to normalize this vector to  $\langle p_1, p_2, \dots, p_N \rangle$  where  $\sum_{i=1}^N p_i = 1$  and  $p_i$  indicates the probability that an input belongs to the  $i$ th class. From now on, with no loss of generality, we assume that the output vector of a DNN is a vector of probabilities as described above.

### 2.2 Neuron Coverage Criteria

To enhance the quality and robustness of DNNs, in the past decade, software engineering researchers have proposed a series of neuron coverage criteria specifically for DNN testing [15, 22, 25, 42]. In this section, we survey the related papers published on peer reviewed venues as follows.

**Neuron Activation Coverage (NAC( $k$ ))** [25]. NAC( $k$ ) is proposed based on the assumption that higher activation coverage implies that more states of a DNN could be explored. The parameter  $k$  of this coverage criterion is defined by users and specifies how a neuron in a DNN can be counted as covered. That is, if the output of a neuron is larger than  $k$ , then this neuron will be counted as covered. The rate of NAC( $k$ ) for a test is defined as the ratio of the number of covered neurons to the total number of neurons.

**k-Multisection Neuron Coverage (KMNC( $k$ ))** [22]. Suppose that the output of a neuron  $o$  is located in an interval  $[low_o, high_o]$ , where  $low_o$  and  $high_o$  are recorded in the training process. To use this coverage criterion, the interval  $[low_o, high_o]$  is divided into  $k$  equal sections, and the goal is to cover all the sections. We say a section is covered by a test if and only if the neuron output is located in the section when the DNN is run against the test. The rate of KMNC( $k$ ) for a test is defined as the ratio of the number of covered sections to the total number of sections. Here, the total number of sections is equal to  $k$  times the total number of neurons.

In most cases, a single test must cover a section in  $[low_o, high_o]$  of each neuron. Only a tiny number of tests do not cover a section in the interval, but cover the boundaries, i.e.,  $(-\infty, low_o]$  and  $[high_o, +\infty)$ . Thus, almost all single tests have the same coverage rate of KMNC( $k$ ).

**Neuron Boundary Coverage (NBC( $k$ ))** [22]. Different from KMNC( $k$ ), NBC( $k$ ) does not aim to cover all sections in  $[low_o, high_o]$ . Instead, it targets to cover the boundaries, i.e.,  $(-\infty, low_o]$  and  $[high_o, +\infty)$ . Using this coverage criterion, we can expect to cover more corner cases. In practice, it is not necessary to directly use  $low_o$  and  $high_o$  as the boundaries. Instead,  $low_o - k\sigma$  and  $high_o + k\sigma$  can be used. Here,  $\sigma$  is the standard deviation of the outputs of a neuron recorded in the training process.  $k$  is a user-defined parameter. The rate of NBC( $k$ ) for a test is defined as the ratio of the number of covered boundaries to the total number of boundaries. Since each neuron has one upper bound and one lower bound, the total number of boundaries is twice the number of neurons.

**Strong Neuron Activation Coverage (SNAC( $k$ ))** [22]. SNAC( $k$ ) can be regarded as a special case of NBC( $k$ ) as it only takes upper boundary into consideration. Thus, it is defined as the ratio of the number of covered upper boundaries to the total number of upper boundaries, in which the latter is actually equal to the number of neurons in a DNN.

**Top- $k$  Neuron Coverage (TKNC( $k$ ))** [22]. TKNC( $k$ ) measures how many neurons have once been the most active  $k$  neurons on each layer. It is defined as the ratio of the total number of top- $k$  neurons on each layer to the total number of neurons in a DNN. We say a neuron is covered by a test if and only if when the DNN is run against the test, the output of the neuron is larger than or equal to the  $k$ th highest value in the layer of the neuron.

It is noteworthy that, according to this definition, this metric only can be used to compare two test sets with more than one test. For a single test, it always covers  $k$  neurons in each layer of a DNN. Thus, TKNC( $k$ ) is always the same for two single tests and, thus, cannot distinguish them.

**Likelihood- and Distance-based Surprise Coverage (LSC( $k$ ) and DSC( $k$ ))** [15]. Surprise coverage relies on the concept of surprise adequacy  $SA(x)$ , which measures the dissimilarity between a test  $x$  and the training data set. The parameter  $k$  here is a pair  $(n, u)$ . Given an upper bound  $u$  and buckets  $B = \{b_1, b_2, \dots, b_n\}$  that divides  $(0, u]$  into  $n$  SA segments, the surprise coverage rate of a set  $X$  of tests is defined as

$$SC(X) = \frac{|\{b_i | \exists x \in X : SA(x) \in (u * \frac{i-1}{n}, u * \frac{i}{n}] \}|}{n}$$

LSC and DSC, two special types of surprise coverage, rely on likelihood-based surprise adequacy (LSA) and distance-based surprise adequacy (DSA), respectively. LSA uses kernel density estimation [41] to estimate the surprise adequacy while DSA uses Euclidean distance. The details on the computation of DSA and LSA are omitted and can be found in the original paper [15].

### 2.3 Coverage-Based Test Prioritization

In conventional software testing, test prioritization (a.k.a., test case prioritization) is actually a classic problem defined by Rothermel et al. [31] as following:

**Test Prioritization.** Given a test set  $T$ , the set  $PT$  of the permutations of  $T$ , and a function  $f$  from  $PT$  to the real numbers, the test prioritization problem is to find  $T' \in PT$  such that

$$\forall T'' \in PT \setminus \{T'\} : f(T') \geq f(T'').$$

Here,  $f(T' \in PT)$  yields an award value for a permutation.

In the past decades, many test prioritization techniques have been proposed for conventional software. Most of these techniques are based on various code coverage information and follow the basic assumption that early maximization of coverage would lead to early detection of faults [10]. Two main coverage-based techniques are known as the coverage-total prioritization and the coverage-additional prioritization [46].

**Coverage-Total Method (CTM).** A CTM is an implementation of the “next best” strategy. It always selects the test with the highest coverage rate, followed by the test with the second-highest coverage rate, and so on. For tests with the same coverage rate, the method will prioritize them randomly. For the example in Table 1, both  $A, B, C, D$  and  $A, B, D, C$  are valid results of CTM.

CTM is attractive because it is relatively efficient and easy to implement. Given a set consisting of  $n$  tests with their coverage rates, CTM only needs to sort these tests according to their coverage rates. Typically, using a quick sort algorithm, it only takes  $O(n \log n)$  time [6].

**Coverage-Additional Method (CAM).** CAM differs from CTM in that it selects the next test according to the feedback from previous selections. It iteratively selects a test that can cover more uncovered code structures. In this manner, we can expect that we can achieve the maximum coverage rate of a test set as soon as possible. After the maximum coverage rate is achieved, we can use CTM to prioritize the remaining unprioritized tests. For the example in Table 1,  $A, D, C, B$  is the only valid result of CAM.

Given a program with  $m$  elements to cover and a set of  $n$  tests, every time we select a test, it will take  $O(mn)$  time to re-adjust the coverage information of the remaining tests. This process will be performed  $O(n)$  times. Thus, the total time cost is  $O(mn^2)$ . According to the time complexity, it is easy to find that CAM is less scalable compared to CTM, especially when  $n$  and  $m$  are very large.

## 3 APPROACH

Owing to the oracle problem discussed before, test prioritization can help label and analyze as many misclassified tests as possible in a limited time. However, due to the problems we argued in Section 1 and as we will show in our evaluation, coverage-based test prioritization becomes ineffective in the context of DNN testing.

**Table 1: An example to illustrate coverage-based test prioritization. ‘X’ means a statement is covered by a test.**

Test	Program Statement							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X				X	X
C	X	X	X	X				
D					X	X	X	X

Therefore, we propose a test prioritization method that is not based on neuron coverage criteria, but based on a statistical view of DNN as discussed in Section 3.1. Such a statistical view inspires us to propose a method, called DeepGini, to prioritize tests of a DNN, which is presented in Section 3.2. Section 3.3 discusses how to improve DNN quality with DeepGini.

### 3.1 A Statistical View of DNN

DNNs are specially good at classifying high-dimensional objects. If we regard each output class of a DNN as a kind of feature of the input object, the computation (or classification) process of a DNN actually maps the original high-dimensional data to only a few kinds of features. As an example, suppose the input of a DNN is a 28x28 image with three channels (i.e., RGB channels). Then the original dimension of the image is  $3^{28 \times 28}$ . In Figure 1, the DNN maps the high-dimension object to a bag (or multi-set)<sup>1</sup>  $B$  of features, in which 94% are features of monkey, 1% are features of tiger, 3% are features of dog, and 2% are features of cat. Since most elements in  $B$  are features of monkey, we classify the input object into the monkey class.

Generally, if the feature bag has the highest purity, i.e., contains only one kind of features (e.g., 100% elements in  $B$  are features of monkey), then there will be no other features confusing our classification and it is more likely that a test input is correctly classified. As an example, in Figure 2, Bag 2 has higher purity than Bag 1. Intuitively, this is because almost all elements in Bag 2 are triangles while Bag 1 has the same number of triangles and circles.

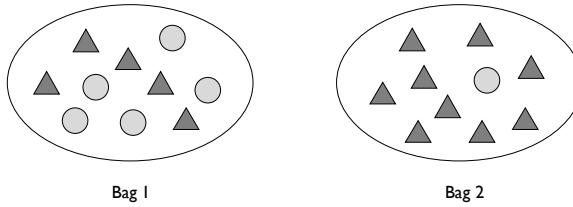
Statistically, if a bag has higher purity, the results of two random samplings in the bag have higher probability to be the same. In contrast, if a bag has lower purity, the results of two random samplings in the bag are more likely to be different. For the example in Figure 2, using sampling with replacement,<sup>2</sup> the probability that two random samplings have the same shape is  $0.5^2 + 0.5^2 = 0.5$  and  $0.1^2 + 0.9^2 = 0.82$  for Bag 1 and Bag 2, respectively.

Formally, assuming the feature distribution in the feature bag output by a DNN is  $\langle p_1, p_2, \dots, p_N \rangle$ , we can compute the probability that two random samplings have different results as  $1 - \sum_{i=1}^N p_i^2$ . The lower the probability, the higher the purity and, thus, the more likely a test input of a DNN is correctly classified.

On the statistical view, we can observe that the problem of measuring the likelihood of misclassification actually has been reduced

<sup>1</sup>A multi-set or a bag is a special kind of set that allows duplicate elements.

<sup>2</sup>In sampling with replacement, after we sample a feature from the feature bag, the feature is put back to the bag so that we have the same probability to get the feature next time.



**Figure 2: Bag 2 has higher purity than Bag 1. Bag 1 has 50% triangles and 50% circles. Bag 2 has 90% triangles and 10% circles.**

to the problem of measuring the purity of a bag. In fact, such a reduction follows the very spirit of the measurement of Gini impurity [26], which inspires us to propose DeepGini for measuring the likelihood of misclassification.

### 3.2 DeepGini: Prioritizing Tests of a DNN

Formally, the metric we use to measure the likelihood of misclassification is defined as below.

**Definition 3.1.** Given a test  $t$  and a DNN that outputs  $\langle p_{t,1}, p_{t,2}, \dots, p_{t,N} \rangle$  ( $\sum_{i=1}^N p_{t,i} = 1$ ), we define  $\xi(t)$  to measure the likelihood of  $t$  being misclassified:

$$\xi(t) = 1 - \sum_{i=1}^N p_{t,i}^2$$

In the definition,  $p_{t,i}$  is the probability that the test  $t$  belongs to the class  $i$ . Figure 3 illustrates the distribution of  $\xi$  when the DNN performs a binary classification. The distribution illustrates that when DNN outputs the same probability for the two classes,  $\xi$  has the maximum value, indicating that we have high probability to incorrectly classify the input test. This result follows our intuition that a test is likely to be misclassified if the DNN outputs similar probabilities for each class, and the rationality of the result has been explained in the previous subsection. The following theorem demonstrates that even though a DNN classifies input tests into more than two classes,  $\xi$  has a similar distribution as that in Figure 3.

**THEOREM 3.2.**  $\xi(t)$  has the unique maximum if and only if  $\forall 1 \leq i, j \leq N : p_{t,i} = p_{t,j}$ .

**PROOF.** According to Lagrangian multiplier method [28], let

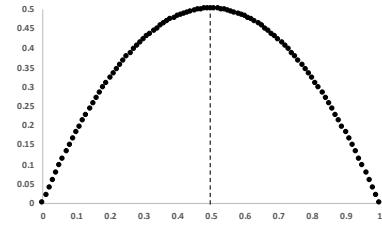
$$L(p_{t,i}, \lambda) = \xi(t) + \lambda \times (\sum_{i=1}^N p_{t,i} - 1)$$

$\forall p_{t,i}$ , let

$$\begin{aligned} \frac{\partial L}{\partial p_{t,1}} &= -2p_{t,1} + \lambda = 0 \\ \frac{\partial L}{\partial p_{t,2}} &= -2p_{t,2} + \lambda = 0 \\ &\vdots \\ \frac{\partial L}{\partial p_{t,N}} &= -2p_{t,N} + \lambda = 0 \end{aligned}$$

If we calculate the difference of any two above equations (e.g. the  $i$ th and  $j$ th equation), we will have

$$2p_{t,i} - 2p_{t,j} = 0 \Rightarrow p_{t,i} = p_{t,j}$$



**Figure 3: Distribution of  $\xi$  for 2-class problem. X-Axis: the probability that a test input belongs to one of the two classes. Y-Axis: the value of  $\xi$ .**

Hence, when  $p_{t,1} = p_{t,2} = \dots = p_{t,N} = 1/N$ ,  $\xi(t)$  has the unique extremum.

At the point  $(p_{t,1}, p_{t,2}, \dots, p_{t,N})$ , the Hessian matrix [1] of  $\xi$  is

$$\begin{bmatrix} -2 & 0 & \dots & 0 \\ 0 & -2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -2 \end{bmatrix}$$

which is a negative definite matrix. This implies that the unique extremum must be the unique maximum [1].  $\square$

We notice that many other metrics such as information entropy [33] also have the above property and is almost equivalent to  $\xi$  [27]. The difference is that it may require a non-statistical view, e.g., the perspective of information theory, to explain the rationality. In addition, we believe that the simplest is the best: the complexity of computing quadratic sum is much easier than that of computing entropy-like metrics because they require logarithmic computation.

According to the above discussion,  $\xi(t_1) > \xi(t_2)$  implies that  $t_1$  is more likely to be misclassified. Hence, to prioritize  $n$  tests in a set, we need to run the tests to collect the outputs, and then sort these tests  $t_i$  according to the value of  $\xi$ .

We argue that the time cost of running the tests is negligible. First, the time cost to run a DNN is not time-consuming like training the DNN. Compared to the expensive cost of manually labeling all tests in a messy order, the time cost is completely negligible. Second, this issue is shared with all neuron-coverage-based test prioritization methods as they also need to run tests to obtain the coverage rates.

**Example 3.3.** Assume that we have four tests  $A, B, C$ , and  $D$  as well as a DNN tries to classify them into three classes. Table 2 shows their output vectors and the values of  $\xi$ . According to the values of  $\xi$ , we can prioritize the tests as  $D, A, C$ , and  $B$ .  $D$  has the highest probability to be misclassified because the DNN outputs the most similar probabilities for each of the three classes. In comparison, for  $B$  and  $C$ , the DNN is more confident about their classes as  $B$  has the probability of 0.8 to be classified into the third class and  $C$  has the probability of 0.6 to be classified into the first class.

Typically, in our prioritization method, we can simply use a quick sort algorithm to sort tests. This algorithm takes  $O(n \log n)$  time complexity. Compared to CTM and CAM, our approach has following merits:

**Table 2: An example to show how DeepGini prioritizes tests.**

Test	Output of DNN	$\xi$
A	$\langle 0.3, 0.5, 0.2 \rangle$	0.62
B	$\langle 0.1, 0.1, 0.8 \rangle$	0.34
C	$\langle 0.6, 0.3, 0.1 \rangle$	0.54
D	$\langle 0.4, 0.4, 0.2 \rangle$	0.64

- The time complexity of our approach is the same with CTM and is much lower than CAM ( $O(mn^2)$ ). Thus, our approach is as scalable as CTM and much more scalable than CAM.
- Different from CTM and CAM, we only need to record output vectors while CTM and CAM require us to profile the whole DNN to record coverage information. Thus, our approach has less interference with the DNN.

### 3.3 Enhancing DNN with DeepGini

Generally, we can add more tests to the training set and retrain the DNN to enhance its robustness. In face of a large number of unlabeled tests and a limited time budget, we cannot label all tests and use them to retrain the DNN. DeepGini allows us to find and label as many misclassified tests as possible in a limited time budget. We observe that the tests prioritized by DeepGini at the front are more effective to improve DNN quality than the tests prioritized at the back. Meanwhile, our empirical study shows that tests prioritized by DeepGini at the front are more effective to improve DNN quality than the tests prioritized at the front but by coverage-based prioritization techniques.

The principle behind the effectiveness of DeepGini actually follows the theory of *active machine learning*, which prefers the tests near the decision boundary (i.e., tests that the DNN is least certain how to label or, equivalently, tests that have the highest value of  $\xi(t)$ ) to actively enhance a deep learning system. We omit the details of active learning here because it is not our contribution. Interested readers can refer to the literature [32] for more details.

To sum up, DeepGini provides not only a test prioritization method but also a technique to enhance the robustness of DNN in a limited time budget.

## 4 EXPERIMENT DESIGN

In this section, we introduce the experimental setup, including the datasets and DNN models we used, approaches to generating adversarial tests, the baseline approaches we compared with, and the research questions we study in the experiments. To conduct the experiments, we implement our approach as well as various neuron-coverage-based test prioritization methods upon Keras 2.1.3 with TensorFlow 1.5.0.<sup>3,4</sup> All of our implementation can be accessed via: <https://github.com/deepgini/deepgini>. All experiments were performed on a Ubuntu 16.04.5 LTS server with one NVIDIA GTX 1080Ti GPU, two 12-core processors “Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz”, and 64GB physical memory.

<sup>3</sup><https://faroit.github.io/keras-docs/2.1.3/>

<sup>4</sup><https://github.com/tensorflow/tensorflow/releases>

### 4.1 Datasets and DNN Models

As shown in Table 3, for evaluation, we select four popular publicly-available datasets, i.e., MNIST,<sup>5</sup> CIFAR-10,<sup>6</sup> Fashion,<sup>7</sup> and SVHN.<sup>8</sup>

The MNIST dataset is for handwritten digits recognition, containing 70,000 input data in total, of which 60,000 are training data and 10,000 are test data.

The CIFAR-10 dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

Fashion is a dataset of Zalando’s article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 gray-scale image, associated with a label from 10 classes.

SVHN is a real-world image dataset that can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images).

To demonstrate the generalizability of our approach, for every data sets, we use two prevalent DNN models in our evaluation. The size of these DNN models ranges from tens to thousands of neurons, exhibiting the diversity of the DNN models to some degree.

### 4.2 Adversarial Test Input Generation.

In addition to prioritizing original tests in the datasets, we also conduct an experiment to prioritize adversarial tests. As the previous study [22], we use four state-of-the-art methods to generate adversarial tests, including FGSM [11], BIM [19], JSMA [24], and CW [5]. These techniques generate tests through different minor perturbations on a given test input. Table 3 shows the total number of adversarial tests generated by these methods in 12 hours.

### 4.3 Baseline Approaches

We compare our approach to coverage-based methods that use eleven different neuron coverage criteria as introduced in Section 2. Since these neuron coverage criteria contain configurable parameters, as shown in Table 4, we use various parameters as suggested by their original authors.

Each comparison experiment is conducted in four modes with regard to two aspects: (1) using CTM or CAM to prioritize tests; and (2) prioritizing tests in the original datasets or prioritizing tests that combine the original tests and the adversarial tests.

### 4.4 Research Questions

DeepGini is designed for facilitating the testers of DNN-based systems to quickly identify misclassified tests and effectively enhance the robustness. Based on this goal, we empirically explore the following three research questions (RQ).

**RQ1. Effectiveness:** Can DeepGini find a better permutation of tests than neuron-coverage-based methods?

We provide answers to RQ1 by computing the values of Average Percentage of Fault-Detection (APFD) metric [46]. Higher APFD values denote faster misclassification-detection rates. When plotting

<sup>5</sup><http://yann.lecun.com/exdb/mnist/>

<sup>6</sup><https://www.cs.toronto.edu/~kriz/cifar.html>

<sup>7</sup><https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/>

<sup>8</sup><http://ufldl.stanford.edu/housenumbers/>

**Table 3: Datasets and DNN models.**

Dataset	Description	DNN Model	#Neurons	#Layers	# Original Tests	# Adversarial Tests
MNIST	Digits 0~9	LeNet-1	42	5	10,000	39,854
		LeNet-5	258	7		39,705
CIFAR-10	Images with 10 classes	ResNet-20	698	20	10,000	40,000
		VGG-16	7242	21		8,000
Fashion	Zalando's article images	LeNet-1	42	5	10,000	39,992
		ResNet-20	698	20		39,905
SVHN	Street view house numbers	LeNet-5	258	7	26,032	104,037
		VGG-16	7242	21		8,000

**Table 4: Configuration parameters for the coverage criteria.**

ID	Criteria	Configuration Parameter $k$		
1	NAC( $k$ )	0	0.75	-
2	KMNC( $k$ )	1,000	10,000	-
3	NBC( $k$ )	0	0.5	1
4	SNAC( $k$ )	0	0.5	1
5	TKNC( $k$ )	1	2	3
6	LSC( $k$ )	(1000, 100)	-	-
7	DSC( $k$ )	(1000, 2)	-	-

the percentage of detected misclassified tests against the number of prioritized tests, APFD can be calculated as the area below the plotted line. It is also noteworthy that although an APFD value ranges from 0 to 1, an APFD value not close to 1 does not mean that the prioritization is ineffective. This is mainly because the theoretically maximal APFD value is usually much smaller than 1 [46]. Formally, for a permutation of  $n$  tests in which there are  $k$  tests will be misclassified, let  $o_i$  be the order of the first test that reveals the  $i$ th misclassified test. The APFD value for this permutation can be calculated as following:

$$APFD = 1 - \frac{\sum_{i=1}^k o_i}{kn} + \frac{1}{2n}$$

To be clear, assuming the theoretical minimum and maximum of the APFD value is *min* and *max*, respectively, we normalize the APFD value from  $[min, max]$  to  $[0, 1]$  so that a prioritization method is better if the APFD value is closer to 1 and is worse if the APFD value is closer to 0.

**RQ2. Efficiency:** Is DeepGini more efficient than neuron-coverage-based methods?

We provide answers to RQ2 by recording the time cost of prioritization. A prioritization method may be very costly because the number of tests is usually very large for a DNN system. According to our evaluation, some prioritization methods cannot finish in several hours, which is not practical in an industrial setting.

**RQ3. Guidance:** Can DeepGini guide the retraining of an DNN to improve its accuracy?

Since the DNNs already have very high accuracy on the original tests ( $> 90\%$  or even  $> 95\%$ ), we cannot clearly show the accuracy improvement of these DNNs. Thus, we leverage the adversarial tests to answer RQ3. For each model, we evenly partition adversarial

test set into a testing set  $T$  and a validation set  $V$  for the following experiment.

After prioritizing the tests in  $T$ , we add back the first 1%, 2%, …, 10% tests into the training set and retrain a new DNN. We do not add more than 50% tests to retrain a DNN because we observe that the accuracy of the DNN will not significantly change with more tests. Using the validation set  $V$ , we compute the accuracy of the new DNN. We repeat the experiment using DeepGini and other coverage metrics and compare the accuracy of the retrained DNNs. According to the experimental results, we answer RQ3 that DeepGini can provide guidance for more effective retraining against the coverage-based methods.

## 5 RESULT ANALYSIS

In this section, we present the results of test prioritization (RQ1 and RQ2) and then analyze whether our approach can better guide the retraining of DNNs (RQ3). Due to the page limit, we cannot present all results in detail. All other results show similar trends and are available online: <https://github.com/deepgini/deepgini>.

### 5.1 Effectiveness and Efficiency (RQ1 & RQ2)

Table 5 lists the results of the LeNet5 net on the MNIST dataset as an example. We summarize our findings in Table 7 and discuss them at the end of this subsection. Based on the feature of these criteria, we compare the results of DeepGini with existing coverage-based methods in two groups: (1) NAC, NBC, and SNAC; (2) TKNC, LSC, DSC, and KMNC.

**5.1.1 Comparing with NAC, NBC, and SNAC.** As shown in Table 5, DeepGini is capable of prioritizing tens of thousands of tests within 2 seconds. The APFD value of DeepGini is very close to 1, which implies that our approach is very close to the theoretically best approach. Table 5 also shows that fewer than 0.5% of tests are sufficient to achieve the maximum coverage rate of the three coverage criteria: NAC, NBC, and SNAC, regardless of their parameter settings. In the 10,000 original tests of MNIST, a very small amount of tests are sufficient for achieving the maximum coverage rate: for NAC(0.75), 22 tests can reach the maximum coverage (84%); for NBC(0.5), 5 tests can reach the maximum coverage (0.97%); and for SNAC(0.5), 5 tests can reach the maximum coverage (2%). Since we achieve the maximum coverage rate very quickly, CAM will degenerate into CTM very quickly. Thus, the effectiveness and the efficiency of CAM are almost the same as CTM for these datasets.

**Table 5: Results of Prioritization (MNIST with LeNet5)**

Metrics	Param.	Original Tests						Original Tests + Adv					
		Max Cov.		CTM		CAM		Max Cov.		CTM		CAM	
		%	#	Time (s)	APFD	Time (s)	APFD	%	#	Time (s)	APFD	Time (s)	APFD
NAC	0	100	1	2	0.638	2	0.638	100	1	11	0.340	11	0.340
	0.75	84	22	2	0.385	4	0.384	86	21	11	0.307	16	0.307
NBC	0	8	38	3	0.638	9	0.638	15	56	14	0.339	40	0.339
	0.5	0.97	5	2	0.638	5	0.637	3	11	13	0.400	20	0.400
	1	0.39	3	3	0.638	6	0.637	2	7	13	0.400	20	0.400
SNAC	0	14	35	3	0.639	9	0.639	22	48	13	0.340	31	0.340
	0.5	2	5	3	0.639	7	0.639	7	11	13	0.340	22	0.340
	1	0.78	3	3	0.638	8	0.638	4	7	13	0.340	20	0.340
TKNC	1	66	86	N/A	N/A	11	0.023	74	96	N/A	N/A	59	0.001
	2	73	67	N/A	N/A	10	0.023	79	70	N/A	N/A	48	0.001
	3	76	57	N/A	N/A	10	0.023	81	55	N/A	N/A	43	0.001
LSC	(1000, 100)	22	220	N/A	N/A	9	0.658	98	982	N/A	N/A	36	0.503
DSC	(1000, 2)	53	531	N/A	N/A	1177	0.658	97	974	N/A	N/A	4738	0.490
KMNC	1000	63	8814	N/A	N/A	34045	0.599	T/O	T/O	N/A	N/A	T/O	T/O
	10000	T/O	T/O	N/A	N/A	T/O	T/O	T/O	T/O	N/A	N/A	T/O	T/O
<b>DeepGini</b>	N/A	N/A	N/A	N/A	N/A	<b>0.45</b>	<b>0.984</b>	N/A	N/A	N/A	N/A	2	<b>0.991</b>

Max Cov.: The maximum coverage rate of the tests (%) and the number of tests to achieve the rate (#).

N/A: Not applicable in theory; T/O: Time out, i.e., cannot get result after running for 12 hours.

**Effectiveness.** Using MNIST as an example, Figure 4 plots the number of detected misclassified tests against the prioritized tests. We have two observations from this figure. First, DeepGini achieves a higher APFD value in comparison to NAC, NBC, and SNAC. Second, as illustrated by the dotted lines in Figure 4, neuron-coverage-based prioritization methods, sometimes, are even worse than the random prioritization strategy.

**Efficiency.** Table 5 shows that, for the original test sets, the CAM-based prioritization processes of NAC, NBC, and SNAC cost at least 2, 5, 7 seconds respectively, while DeepGini costs only 0.45 seconds. Similarly, for the test set with the adversarial examples, we observe that the CAM-based prioritization processes of the three baselines cost more than 11 seconds, while DeepGini costs only 2 seconds. This data shows that DeepGini has a higher efficiency in comparison with NAC, NBC, and SNAC.

**5.1.2 Comparing with TKNC, LSC, DSC, and KMNC.** As discussed in Section 2.2, every single test has the same coverage rate of TKNC, LSC, and DSC, regardless of its parameter  $k$ . Thus, CTM does not work if we use these coverage metrics to prioritize tests. Unfortunately, CAM does not work well using these coverage metrics either. The main reason is that fewer than 5% of tests are enough to achieve the maximal coverage rate. After prioritizing the 5% tests, CAM is degenerate into CTM, which does not work as explained above.

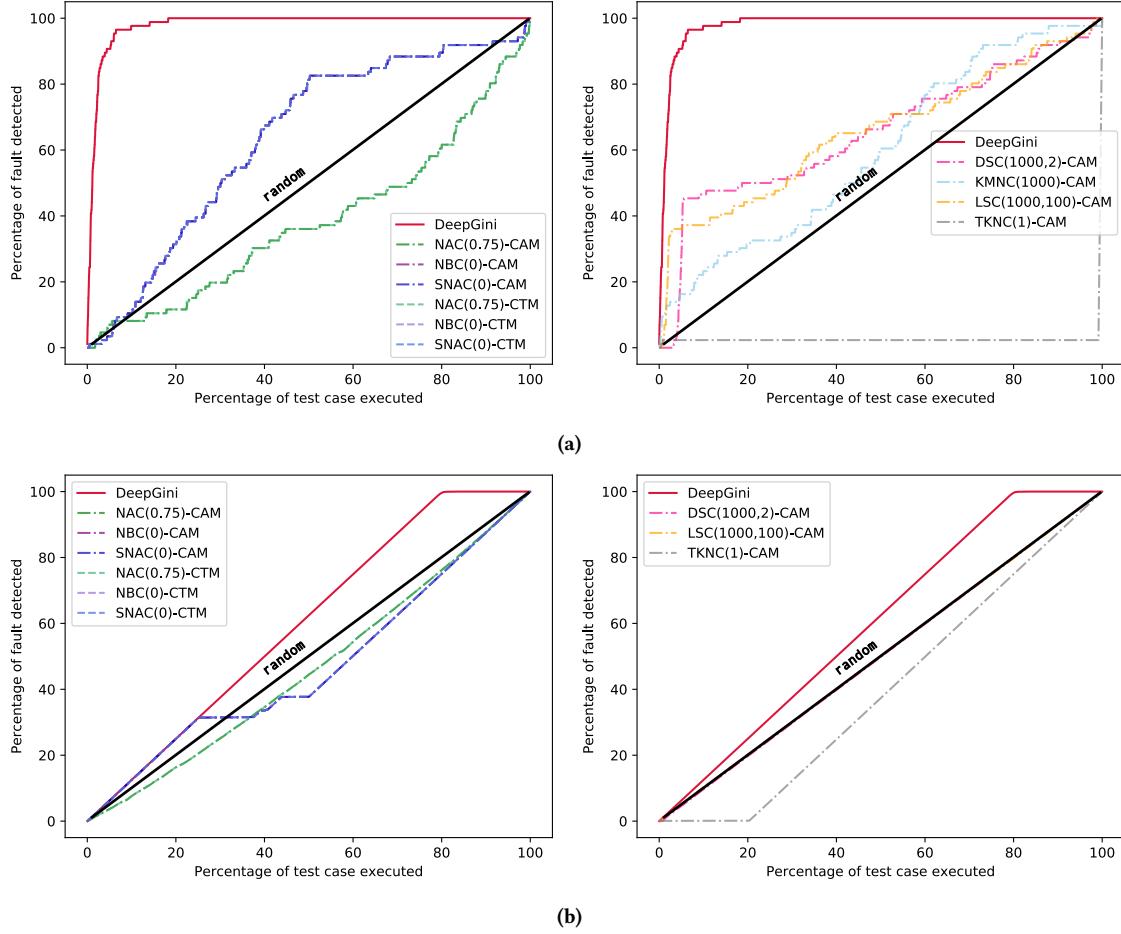
Similarly, CTM does not work if we use KMNC to prioritize tests, because almost all single tests have the same coverage rate of KMNC, regardless of its parameter  $k$ . However, KMNC can work with the CAM prioritization method. Thus, we only compare KMNC-based CAM with our prioritization method.

**Effectiveness.** In the example of LeNet-5 on MNIST, Figure 4 plots the prioritization results, in which the curve of our method goes up far more quickly than the four baseline methods. In the original test set, while DeepGini has obtained the APFD value of 0.984, TKNC, LSC, DSC, and KMNC only obtain 0.023, 0.658, 0.658, and 0.599. We can observe similar trends for the test set with adversarial examples. This result implies that the DeepGini significantly outperforms the four baselines regarding the effectiveness of prioritizing tests.

**Efficiency.** Table 5 shows that prioritization methods based on these coverage metrics are 20×–2000× slower than our method. One special case is KMNC-based CAM method. When prioritizing tests using KMNC-based CAM method, we observe serious efficiency issues. That is because the time complexity of KMNC-based CAM method is very high, we cannot finish prioritizing tests in an acceptable time budget. For MNIST, we cannot succeed in prioritizing tests using the method in 12 hours. Considering MNIST is a relatively small dataset, the efficiency problem would be the bottleneck of applying KMNC-based CAM method in practice.

## 5.2 Guidance (RQ3)

Figure 5 illustrates the experimental results for RQ3. Like the previous experiments, we put the coverage metrics into two groups. Figure 5-a and Figure 5-b demonstrate the results of LeNet-5 on MNIST. The curves show the accuracy of the DNN after retraining with 1%, 2%, ..., 10% tests. As we introduced in Section 4.4, the testing set  $T$  and validation set  $V$  is divided from the adversarial test data. This setting makes initial accuracy of the DNN are the same for all metrics, i.e., it is 0 without retraining. Note that, because



**Figure 4: Test prioritization for MNIST with LeNet5. X-Axis: the percentage of prioritized tests (sub-figure a), or percentage of both original and adversarial tests (sub-figure b); Y-Axis: the percentage of detected misclassified tests.**

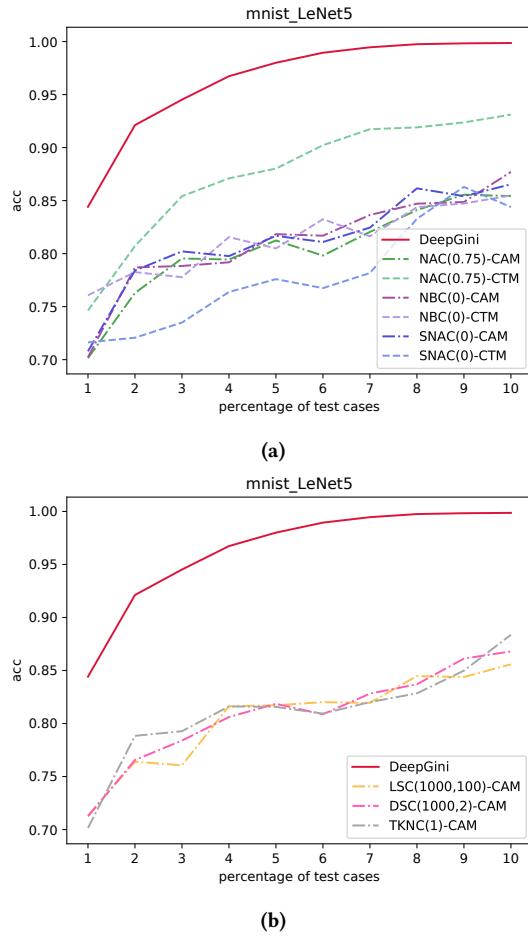
**Table 6: The DNNs' accuracy value after retraining with first 10% prioritized tests.**

	MNIST		CIFAR-10		FASHION		SVHN		Avg
	LeNet-1	LeNet-5	ResNet-20	VGG-16	LeNet-1	ResNet-20	LeNet-5	VGG-16	
NAC(0.75)-CTM	0.89	0.93	0.93	0.79	0.81	0.91	0.82	0.67	0.84
NAC(0.75)-CAM	0.83	0.85	0.92	0.76	0.78	0.94	0.8	0.74	0.83
NBC(0)-CAM	0.84	0.88	0.92	0.78	0.77	0.94	0.81	0.75	0.84
NBC(0)-CTM	0.91	0.84	0.93	0.75	0.81	0.95	0.82	0.75	0.85
SNAC(0)-CTM	0.84	0.84	0.93	0.79	0.83	0.95	0.82	0.75	0.84
SNAC(0)-CAM	0.82	0.87	0.91	0.76	0.79	0.94	0.81	0.76	0.83
LSC(1000, 100)-CAM	0.84	0.86	0.92	0.81	0.8	0.94	0.81	0.74	0.84
DSC(1000, 100)-CAM	0.84	87	0.92	0.82	0.81	0.94	0.82	0.75	0.85
TKNC(1)-CAM	0.83	0.88	0.92	0.8	0.78	0.94	0.81	0.8	0.85
DeepGini	<b>0.99</b>	<b>1</b>	<b>0.98</b>	<b>0.93</b>	<b>0.89</b>	<b>0.98</b>	<b>0.93</b>	<b>0.97</b>	<b>0.96</b>

KMNC-based prioritization technique is not scalable, we cannot finish the experiment of RQ3 in 12 hours.

The curves show that retraining with the tests prioritized by DeepGini is more effective in improving the accuracy of DNNs. This outperformance is clear in the retraining with all sizes of the test sets. We also present the accuracy value after retraining

the DNN with the first 10% prioritized tests in Table 6. From the table, we can observe that the outperformance can be found across all combinations of datasets and DNN models. In average, while the baseline criteria can reach 0.83 – 0.85 accuracy, DeepGini can improve the accuracy value to 0.96, which is close to 1.



**Figure 5: Enhancing the robustness of the DNN with prioritized tests (MNIST with LeNet-5).**

### 5.3 Discussion

For each of these existing neural coverage based criteria, we summarize our findings in Table 7. Although structural coverage criteria are very effective in classic testing methods, they are not effective in the new scenario of DNN testing. More specifically, for some of these criteria, we observe that a very small part, around 1% to 5%, of original tests are sufficient to achieve the maximal coverage. For some of these criteria, the coverage criteria cannot distinguish different tests. These results extend the existing discussion on the effectiveness of structural coverage [21] – some researchers cast doubts on the effectiveness of neuron-coverage-guided test generation techniques. For instance, in our experiment, the random prioritization strategy even outperforms NAC and TKNC, which implies that these coverage criteria could be misleading in finding new incorrect DNN behaviors.

Further, DeepGini is designed based on the assumption that the test produces similar probabilities for all classes has a higher probability of being misclassified. Admittedly, this assumption is not necessarily true. There could be some variance/difference in the probabilities of the different classes, yet a wrong class may be given

a higher probability than the expected class. However, the results of RQ1 shows the APFD value of DeepGini is higher than baselines, which indicates that it can efficiently detect a large number (but not all) of misclassified tests in comparison with baselines.

On the other hand, test prioritization should be scalable and efficient so that it can be applied in the scenario of DNN testing. Such scalability and efficiency requirement are very necessary because the number of tests is usually very large in DNN testing, which is different from conventional software testing. For instance, we found that the KMNC criteria failed to work well regarding both the effectiveness and efficiency. In the future, more research should be conducted to investigate its potential and improvement. Considering MNIST is a very basic dataset for deep learning, we suggest software engineers do not apply these criteria in their engineering practice before they are improved.

## 6 RELATED WORK

We discuss the related work in two groups: (1) test prioritization methods for conventional software and (2) testing techniques for deep learning systems.

### 6.1 Test Prioritization Techniques

Test prioritization seeks to find the ideal ordering of tests, so that software testers or developers can obtain maximal benefit in a limited time budget. The idea was first mentioned by Wong et al. [44] and then the technique was proposed by Harrold and Rothermel [12, 29] in a more general context. We observe that such an idea from the area of software engineering can significantly reduce the effort of labeling for deep learning systems. This is mainly because a deep learning system usually has a large number of unlabeled tests but developers only have limited time for labeling.

Coverage-based test prioritization, such as the CAM and CTM methods studied in this paper, is one of the most commonly studied prioritization techniques. In conventional software engineering, we can obtain a new prioritization method when a different coverage criterion is applied. Rothermel et al. [30, 31] reported empirical studies of several coverage-based approaches, driven by branch coverage, statement coverage, and so-called FEP, a coverage criterion inspired by mutation testing [3]. In addition, Jones and Harrold [14] reported that MC/DC, a stricter form of branch coverage, is also applicable to coverage-based test prioritization. Different from the above techniques, we focus on testing and debugging for deep learning systems. Thus, we studied test prioritization based on coverage criteria that specially proposed for DNNs. Our study demonstrated that, using these coverage criteria, coverage-based test prioritization is not effective and efficient. Sometimes, its effectiveness is even worse than random prioritization. Instead, our approach uses a simple metric that does not require to profile the DNNs but is effective and also efficient.

We notice that, in software engineering, there are also many prioritization techniques based on metrics other than coverage criteria, including distribution-based approach [20], human-based approach [40, 47], history-based approach [34], model-based approach [16–18], and so on. These techniques are specially-designed for conventional software systems instead of deep learning systems.

**Table 7: Summary of Our Findings on Test Prioritization**

Metrics	Findings
NAC/NBC/SNAC	1. CAM will quickly degenerate into CTM for these metrics because only a small number of tests can achieve the maximum coverage. 2. CTM is not effective and even worse than random prioritization when we use these metrics.
TKNC/LSC/DSC	3. CAM will quickly degenerate into CTM for TKNC/LSC/DSC because only a small number of tests can achieve the maximum coverage rate. 4. CTM does not work when TKNC/LSC/DSC are used because all single tests have the same coverage rate. 5. Computing LSC/DSC additionally relies on the training set.
KMNC	6. CAM is not scalable due to its high complexity when KMNC is used. 7. CTM does not work when KMNC is used because almost all single tests have the same coverage rate.
DeepGini	8. DeepGini is the most effective and efficient metric for test prioritization regarding the APFD value and the time cost. 9. DeepGini does not relies on anything except for the tests and the DNN to test.

Making them applicable to deep learning systems may require non-trivial efforts of re-design. We leave them as our future work.

## 6.2 Testing Deep Learning Systems

In conventional practice, machine learning models were mainly evaluated using available validation datasets [43]. However, these datasets usually cannot cover various corner cases that may induce unexpected behaviors [25, 39]. To further ensure the quality of a deep learning system, software-engineering researchers have designed many testing approaches. Pei et al. [25] proposed *DeepXplore*, the first white-box testing framework, to identify and generate the corner-case inputs that may induce different behaviors over multiple DNNs. Ma et al. [23] presented a mutation testing framework for DNNs aiming at evaluating the quality of datasets. Tian et al. [39] presented *DeepTest* to generate test inputs by maximizing the numbers of activated neurons via a basic set of image transformations. Zhang et al. [49] employed generative adversarial network to transform the driving scenes into various weather conditions, which increases the diversity of datasets. Different from the above techniques that rely on solid test oracle, our method focuses on the problem that we usually have a large number of tests without test oracle. We observe that the idea of test prioritization can enable developers to obtain as many misclassified tests as possible in a limited time budget, thereby easing the burden of labeling. However, in comparison with traditional software programs, the modern DNNs often consist of millions of neurons and hundreds of layers, which naturally enlarges its potential testing space. While the sophisticated internal logic of a DNN makes it challenging to adopt the idea of coverage criteria to test prioritization, this paper introduces a new metric that only analyzes the output space of a DNN and is able to effectively guide the test prioritization. We notice that researchers have proposed some preliminary prioritization methods for testing DNNs [4, 48]. They use different methods to prioritize the tests but failed to compare their techniques with classic coverage-based methods. Furthermore, they did not provide any information on the capability of enhancing the DNN robustness.

To guide the testing techniques for DNNs, Pei et al. [25] introduced neuron activation coverage to measure the differences of the execution of test data. Ma et al. [22] designed a set of multi-level and multi-granularity testing criteria for assessing the quality of testing of deep learning systems. Our approach has shown that it is not effective or efficient to prioritize tests based on these coverage

criteria. Sun et al. [37, 38] presented a concolic testing framework that incrementally generates a set of test inputs to improve coverage by alternating between concrete execution and symbolic analysis. The MC/DC-like coverage criteria proposed in the paper [37, 38] does not work for prioritization because of two reasons. First, since we need at least a pair of tests to compute the coverage rate, the coverage rate of a single test is meaningless. Thus, CTM does not work. Second, computing the coverage rate is of at least quadratic complexity. Thus, it is not scalable, just like the KMNC coverage shown in our evaluation. Since we only focus on coverage criteria published in peer-reviewed venues, these MC/DC-like metrics are not included but just briefly discussed here. Different from such test generation techniques that also have the oracle problem, our approach attempts to prioritize tests so that the oracle problem is alleviated.

## 7 CONCLUSION

Based on a statistical view of DNN, we have introduced an approach, namely DeepGini, to prioritizing testing data so that we can improve the quality of DNN efficiently. Experimental results demonstrate that it is more effective and efficient than coverage-based methods. In real-world scenario, tests usually do not have labels and we have to invest a lot of manpower to label them. With such a prioritization method in hand, we can achieve maximal benefit, even the labeling process is prematurely halted at some arbitrary point due to resource limits.

## ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their insightful comments. This project was partially funded by the National Natural Science Foundation of China under Grant Nos. 61832009 and 61932012. Qingkai Shi is the corresponding author.

## REFERENCES

- [1] Ken Binmore and Joan Davies. 2002. *Calculus: concepts and methods*. Cambridge University Press.
- [2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).
- [3] Timothy Alan Budd. 1981. Mutation Analysis of Program Test Data. (1981).
- [4] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. 2019. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 63–70.

- [5] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–57.
- [6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.
- [7] Alex Davies. [n. d.]. Tesla’s Latest Autopilot Death Looks Just Like a Prior Crash. Available at <https://www.wired.com/story/teslas-latest-autopilot-death-looks-like-prior-crash/> (2020/01/27). ([n. d.]).
- [8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
- [9] John S Denker and Yann LeCun. 1991. Transforming neural-net output levels to probability distributions. In *Advances in neural information processing systems*. 853–859.
- [10] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2013. Coverage-based test case prioritization: An industrial case study. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 302–311.
- [11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *Proceedings of 2015 3rd International Conference on Learning Representations (ICLR)*.
- [12] Mary Jean Harrold. 1999. Testing evolving software. *Journal of Systems and Software* 47, 2-3 (1999), 173–181.
- [13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [14] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering* 29, 3 (2003), 195–209.
- [15] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE ’19)*. IEEE Press, 1039–1049.
- [16] Bogdan Korel, George Koutsogiannis, and Luay H Tahat. 2007. Model-based test prioritization heuristic methods and their evaluation. In *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM, 34–43.
- [17] Bogdan Korel, George Koutsogiannis, and Luay H Tahat. 2008. Application of system models in regression test suite prioritization. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 247–256.
- [18] Bogdan Korel, Luay Ho Tahat, and Mark Harman. 2005. Test prioritization using system models. In *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*. IEEE, 559–568.
- [19] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2017. Adversarial Examples in the Physical World. In *Proceedings of 2017 5th International Conference on Learning Representations (ICLR)*.
- [20] David Leon and Andy Podgurski. 2003. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *2003 IEEE 14th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 442.
- [21] Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. 2019. Structural coverage criteria for neural networks could be misleading. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 89–92.
- [22] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 120–131.
- [23] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.
- [24] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 372–387.
- [25] Kexin Pei, Yinzi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.
- [26] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.
- [27] Laura Elena Raileanu and Kilian Stoffel. 2004. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence* 41, 1 (2004), 77–93.
- [28] R Tyrrell Rockafellar. 1993. Lagrange multipliers and optimality. *SIAM review* 35, 2 (1993), 183–238.
- [29] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.
- [30] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM’99) Proceedings. IEEE International Conference on*. IEEE, 179–188.
- [31] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.
- [32] Burr Settles. 2009. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.
- [33] Claude Elwood Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948), 379–423.
- [34] Mark Sheriff, Mike Lake, and Laurie Williams. 2007. Prioritization of regression tests using singular value decomposition with empirical change records. In *Software Reliability, 2007. ISSRE’07. The 18th IEEE International Symposium on*. IEEE, 81–90.
- [35] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.
- [36] Jack Stewart. [n. d.]. Tesla’s Autopilot Was Involved in Another Deadly Car Crash. Available at <https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/> (2020/01/27). ([n. d.]).
- [37] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *arXiv preprint arXiv:1803.04792* (2018).
- [38] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. *arXiv preprint arXiv:1805.00089* (2018).
- [39] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.
- [40] Paolo Tonella, Paola Avesani, and Angelo Susi. 2006. Using the case-based ranking methodology for test case prioritization. In *Software Maintenance, 2006. ICSM’06. 22nd IEEE International Conference on*. IEEE, 123–133.
- [41] Matt P Wand and M Chris Jones. [n. d.]. *Kernel Smoothing*. CRC Press.
- [42] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 408–426.
- [43] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.
- [44] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. 1998. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience* 28, 4 (1998), 347–369.
- [45] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2016. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256* (2016).
- [46] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.
- [47] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 201–212.
- [48] Long Zhang, Xuechao Sun, Yong Li, and Zhenyu Zhang. 2019. A noise-sensitivity-analysis-based test prioritization technique for deep neural networks. *arXiv preprint arXiv:1901.00054* (2019).
- [49] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 132–142.
- [50] Chris Ziegler. [n. d.]. A Google self-driving car caused a crash for the first time. Available at <https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report> (2020/01/27). ([n. d.]).



# Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries

Saeid Tizpaz-Niari

University of Colorado Boulder  
Boulder, CO, USA  
saeid.tizpazniari@colorado.edu

Pavol Černý

TU Wien  
Vienna, Austria  
pavol.cerny@tuwien.ac.at

Ashutosh Trivedi

University of Colorado Boulder  
Boulder, CO, USA  
ashutosh.trivedi@colorado.edu

## ABSTRACT

Programming errors that degrade the performance of systems are widespread, yet there is very little tool support for finding and diagnosing these bugs. We present a method and a tool based on *differential performance analysis* – we find inputs for which the performance varies widely, despite having the same size. To ensure that the differences in the performance are robust (i.e. hold also for large inputs), we compare the performance of not only single inputs, but of classes of inputs, where each class has similar inputs parameterized by their size. Thus, each class is represented by a performance function from the input size to performance. Importantly, we also provide an explanation for why the performance differs in a form that can be readily used to fix a performance bug.

The two main phases in our method are discovery with fuzzing and explanation with decision tree classifiers, each of which is supported by clustering. First, we propose an evolutionary fuzzing algorithm to generate inputs that characterize different performance functions. For this fuzzing task, the unique challenge is that we not only need the input class with the worst performance, but rather a set of classes exhibiting differential performance. We use clustering to merge similar input classes which significantly improves the efficiency of our fuzzer. Second, we explain the differential performance in terms of program inputs and internals (e.g., methods and conditions). We adapt discriminant learning approaches with clustering and decision trees to localize suspicious code regions.

We applied our techniques on a set of micro-benchmarks and real-world machine learning libraries. On a set of micro-benchmarks, we show that our approach outperforms state-of-the-art fuzzers in finding inputs to characterize differential performance. On a set of case-studies, we discover and explain multiple performance bugs in popular machine learning frameworks, for instance in implementations of logistic regression in scikit-learn. Four of these bugs, reported first in this paper, have since been fixed by the developers.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404540>

## KEYWORDS

Differential Performance Bugs, ML Libraries, Testing, Debugging

### ACM Reference Format:

Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2020. Detecting and Understanding Real-World Differential Performance Bugs in Machine Learning Libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395363.3404540>

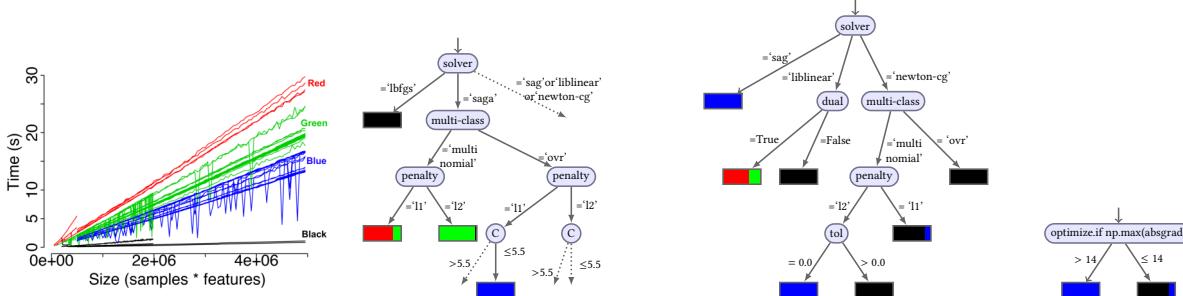
## 1 INTRODUCTION

The defects in software developments can lead to severe performance degradations and waste valuable system resources such as CPU cycles. Moreover, studies have shown that such performance bugs are widespread in real-world applications [13, 36, 38]. How can a user of a program recognize a performance bug? Most often, they can suspect a bug if the program has different performance for similar inputs. Recently, such bugs have been reported by users of machine learning libraries [18, 19, 27, 37]. For example, a user of random forest regression reported that the trees with ‘mae’ criterion are slower than those with ‘mse’. Once such an issue is discovered, how can a maintainer of a library know whether the difference is the result of a bug, or if it is inherent in the problem being solved?

We present a method and a tool to address the challenges we just described. We use *fuzzing* to generate inputs that uncover performance bugs and *discriminant analysis* to explain the differences in performance. The fuzzing part of our study generalizes evolutionary-based fuzzing algorithms [1] in two ways. *First, we consider multiple populations of inputs.* Each population corresponds to a *simple path*<sup>1</sup> in the program’s control-flow graph. An execution that takes a loop 5 times is represented by the same simple path as an execution that takes the loop 7 times. Therefore, each class (or population) of inputs is represented by a function from the input size to performance. *Second, rather than searching for the single worst-case input class, our fuzzer explores classes of inputs with significant performance differences.* To do so, we propose an evolutionary algorithm such that it both models the performance as a function of input size and finds a set of classes of inputs with diverse performance efficiently. The key idea for the efficiency is to combine evolutionary fuzzing algorithms with the functional data clustering [6]. The clustering is used to focus the search to retain representative paths from a few prominent clusters of paths instead of repeatedly exploring paths with similar performance.

---

<sup>1</sup>A path is simple if it contains an edge at most once.



**Figure 1:** From left to right: (a) four classes of performances discovered by fuzzing and clustering, (b) decision tree learned in the space of parameter features (part 1), (c) decision tree learned in the space of parameter features (part 2), (d) decision tree learned using the internal features to explain differences in performance of ‘newton-cg’ solver.

Once a suspicious performance abnormality has been uncovered, the next step is to pinpoint the root causes for the differential performance. For this task, we adapt techniques from discriminant learning algorithms [2, 38] to functional data [17]. The causes of the diverse performance are explained using features from the space of program (hyper)parameters and from the space of program internals (such as the number of invocations of a particular method). We learn a discriminant model that shows what features are the same inside a cluster and what features distinguish one from another.

Previous works in performance fuzzing consider the worst-case algorithmic complexity [8, 16] and search for a single input with the significant resource usage. Differential fuzzing is used in security to detect timing side channels in Java applications [11, 40]. To the best of our knowledge, this is the first work to automatically generate inputs to characterize multiple performance classes. Also, this work utilizes the inputs from the fuzzer to automatically find code regions contributing to the differential performance, while previous works focusing on performance bug localization assume that the interesting inputs are given [36, 38].

We apply our approach on a set of micro-benchmark programs and larger machine learning libraries. On a set of micro-benchmarks that include well-known sorting, searching, tree, and graph algorithms [35], we demonstrate that although our approach is slower in generating inputs, it outperforms other fuzzing techniques [8, 16] in characterizing differential performance. On a set of eight larger machine learning tools and libraries, we find multiple previously-unreported performance bugs in widely used libraries such as in the implementation of logistic regression in scikit-learn framework [15]. We have reported these bugs, and four of them have since been fixed by the developers.

The key contributions of our paper are:

- We extend fuzzing algorithms to functional data and, crucially, use clustering *during* the fuzzing process to efficiently find classes of inputs with widely different performance. We show the importance of clustering during the fuzzing phase through comparison to state-of-the-art fuzzers.
- We use the discriminant learning approach alongside the fuzzing to find the root cause of performance issues.
- We implement our approach in the tool DPFuzz and evaluate it on eight machine learning libraries. We show the

usefulness of DPFuzz in finding and explaining multiple performance bugs such as in scikit-learn libraries [15].

## 2 OVERVIEW

First, we show how DPFuzz can be used to detect a performance bug in a popular machine learning library. Then, we describe the components of DPFuzz.

**A) Applying DPFuzz on Logistic Regression.** Logistic regression in scikit-learn [15] is a popular classification model that supports various solvers and penalty functions. We refer the reader to [24] for more information about the functionality of this classifier. We analyzed the performance of logistic regression<sup>2</sup>.

**Task.** We apply DPFuzz to automatically generate inputs that find diverse classes of performances. If there are multiple classes (more than one cluster), we explain the performance issues with DPFuzz.

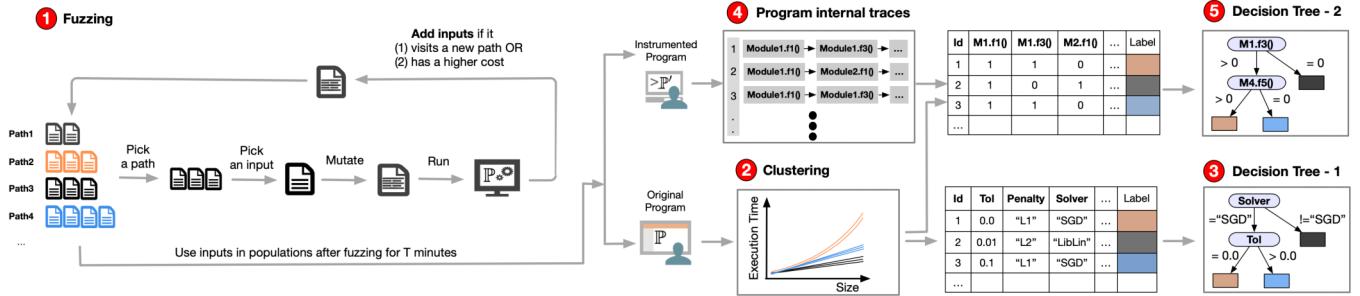
**Generating interesting inputs.** We run the fuzzer for about two hours, and it generates 185 sets of inputs, where each input corresponds to a unique path in the control flow graph (CFG) of the program. Since each set includes multiple inputs, the performance of each path gives rise to functions. The fuzzer thus provides 185 different performance functions. The coverage of DPFuzz is 78% where it covers 2.1K LoC from almost 2.7K LoC.

**Clustering performance functions.** Given the performance functions, we use functional data clustering [6] algorithms to group them into a smaller number of clusters. Figure 1 (a) shows that the performance functions are clustered into 4 groups.

**Explaining the clusters of performances.** Given the set of inputs and their performance labels, we use the discriminant learning approach [39] with decision tree algorithms to explain the differences between performance clusters in terms of program hyperparameters and program internal features.

The decision trees in Figure 1 (b) and (c) show the discriminant models in the space of hyper-parameters. One particular (expected) observation is that different solvers have different performance. However, more interesting parts are those with the differential performance in the same solver. For example, Figure 1 (c) shows the inputs for ‘newton-cg’ solver can be in black (fast) or blue (slow)

<sup>2</sup>[https://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.LogisticRegression.html](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)



**Figure 2:** The workflow of DPFuzz. The three main tasks are (1) fuzzing to generate interesting inputs, (2) clustering to find classes of performance functions, and (3) explaining different performance classes with decision tree-1 in the space of program inputs and decision tree-2 in the space of program internals.

clusters. In particular, there is an unexpected performance differences between ‘tol’ = 0.0 and ‘tol’ > 0.0 (even for ‘tol’ = 0.000001). The inputs with ‘tol’ = 0.0 follow (slow) blue cluster whereas the inputs with positive values follow (fast) black cluster.

The next step is to help the library maintainer explain the differences between the fast and slow clusters in terms of library internals such as function calls. DPFuzz obtains 1,673 features about the internals of logistic regression through instrumentations. The decision tree model in Figure 1 (d) shows the discriminant model for ‘newton-cg’ solver in the space of the internal features (chosen from a set of 5 accurate models). The model shows that the number of calls to ‘if np.max(absgrad)’ in the optimize module is the discriminant that distinguishes the blue and black clusters:

```

while k < maxiter:
    # call to compute gradients
    absgrad = np.abs(fgrad)
    if np.max(absgrad) < tol:
        break
    # inner loop: call to compute loss

```

The outer loop of Newton iterations will be unnecessary taken even if np.max(absgrad) becomes 0.0 for the case where the tolerance (‘tol’) is set to zero. This wastes CPU resources by calling to compute the gradient, Hessian, and loss unnecessarily. We mark this as performance bug ①. With this error localization, the library maintainer can see that the fix is to replace the strict inequality with a non-strict one. We reported this bug [29], and the developers have confirmed and fixed it using this information [32].

Another interesting performance issue in logistic regression is related to ‘saga’ solver. DPFuzz automatically found the issues in the solver that was already reported in the issue database [27]. Figure 1 (b) shows the discriminant learned for ‘saga’ solver.

**Comparison with the existing fuzzers.** The existing performance fuzzers such as SlowFuzz [16] and PerfFuzz [8] are looking for the worst-case execution time, and it makes them unlikely to find differential performance bugs. In Figure 1 (a), the worst-case behavior (the red function) is not related to the bug found by DPFuzz. The performance bug ① is found because of the differences between blue and black functions, and neither of them is the worst-case behavior. Exploring and explaining various classes of performance are the novelty in DPFuzz that find these subtle performance bugs.

**B) Inside DPFuzz** Figure 2 shows different components of DPFuzz. The *first* component of DPFuzz is to generate inputs. For this part, we extend the evolutionary fuzzing algorithms [1, 9] with functional data analysis and clustering [6, 17]. Our fuzzing approach considers multiple populations (one population per a distinct path in the CFG). Then, it picks a cluster, a path from the cluster, and an input from the selected path. Next, it mutates and crossovers the input and runs the input on the target program. This returns the cost of executions (either in terms of actual execution times or in terms of executed lines), and the path characterization. The fuzzing approach adds a new input to the populations if the input has visited a new path in the program or the input has achieved higher costs in comparison to the inputs in the same simple path. The fuzzing stops after  $T$  time units and provides generated inputs for debugging.

The *second* component of DPFuzz is to characterize different performance classes. The set of inputs in a path (or a population) defines a performance function varied in the input size. Given  $n$  paths (corresponds to  $n$  performance functions), DPFuzz applies clustering algorithms to partition these functions into  $k$  classes of performances ( $k \leq n$ ). The clustering is primarily based on the non-parametric functional data clustering [6] with  $l_1$  distance. The clustering finds similar input classes and separates classes with significant performance differences. The plot in Figure 1 (a) is generated as a result of fuzzing and clustering steps.

The *third* component of DPFuzz is to explain the differential performance in terms of program inputs and internals. The CART decision tree inference [2] is mainly used to obtain the explanation models. In the space of program inputs, the features are input parameters such as the value of “solver”, and the labels are the performance classes from the clustering algorithm. The decision tree-1 in Figure 2 is a sample model in the space of program inputs. The model shows what input parameters are common in the same cluster and what the parameters distinguish different clusters. The models in Figure 1 (b) and (c) are produced from this step. Using this decision tree, the user may realize that all or some aspects of differential performance are unexpected. The idea is to find code regions that contribute to the creation of an unexpected performance.

In the space of program internals, the instrumentation of target programs is used to obtain program internal traces. For this aim, tracing techniques [14] are applied to generate a trace of execution

for inputs from either the whole population or relevant to the unexpected performance. Next, we gather program internal features from these traces. The features used in this work are the number of calls to functions, conditions, and loops. Given these program internal features and the performance class labels (from the clustering algorithm), the problem of localizing code regions (related to the differential performance) becomes a standard classification problem. The decision tree-2 in Figure 2 is learned in the space of program internal features that show what properties of program internals are most likely responsible for differential performance. Figure 1 (d) is produced from this step.

### 3 PROBLEM STATEMENT

Following the work of Hartmanis and Stearns [5], it is customary to characterize the resource complexity of a program as a function of the program input size characterizing the worst/average/best performance. However, often there are latent modes in the program inputs characterizing widely different complexity classes, and knowing the existence and explanation of their differences will serve as a debugging aid for the developers and users. We study the problem of discovery (via evolutionary fuzzing) and explanation (via classifications) of latent resource complexity classes. To formalize our problem, we use the following abstract model:

**DEFINITION 3.1 (PERFORMANCE ABSTRACTION OF A PROGRAM).** An abstract performance model  $\llbracket \mathcal{P} \rrbracket$  of a program  $\mathcal{P}$  is a tuple  $\llbracket \mathcal{P} \rrbracket = (X, Y, O, \pi)$  where:

- $X = \{x_1, \dots, x_n\}$  is the set of input variables characterizing the input space  $\mathcal{D}_X$  of the program,
- $Y = \{y_1, y_2, \dots, y_m\}$  is the set of trace variables characterizing the execution space  $\mathcal{D}_Y$  of the program,
- $O : \mathcal{D}_X \rightarrow \mathcal{D}_Y$  is the functional output of the program summarizing effect of the input on trace variables, and
- $\pi : \mathcal{D}_X \rightarrow \mathbb{R}_{\geq 0}$  is the performance (running time or memory usage) of the program.

Following the asymptotic resource complexity convention, we assume the existence of a function  $|\cdot| : \mathcal{D}_X \rightarrow \mathbb{N}$  providing an estimate of the input size. For ML applications, the size can be a product of number of samples and features. For a subset  $S \subseteq \mathcal{D}_X$  of the input space, we define its performance class as the function  $\pi_S : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  defined as worst-case complexity given below:

$$\pi_S \stackrel{\text{def}}{=} n \mapsto \sup_{x \in S : |x|=n} \pi(x).$$

Note that  $\pi_S$  is a partial function for finite sets  $S \subset \mathcal{D}_X$ . Let  $\Pi$  be the set of all performance classes. We introduce the distance function  $d_p : \Pi \times \Pi \rightarrow \mathbb{R}_{\geq 0}$  on the set of performance classes parameterized with  $p$ . The typical choice for  $d_p$  is  $p$ -norm ( $p = 1, 2, \infty$ ) and  $R^2$  coefficient of determinations.

**DEFINITION 3.2 (DIFFERENTIAL PERFORMANCE FUZZING).** Given a program  $\mathcal{P} = (X, Y, O, \pi)$  and a separation bound  $\varsigma > 0$ , the differential performance fuzzing problem is to find a set partition  $P = \{D_1, D_2, \dots, D_k\}$  of  $\mathcal{D}_X$  such that for every  $A, B \in P$  we have that  $d_p(\pi_A, \pi_B) > \varsigma$ .

Even when the input space is finite, the differential performance fuzzing problem requires an exhaustive search in the exponential

set of subsets of input space, and hence is clearly intractable. In Section 4.1, we present an evolutionary algorithm to solve the problem by restricting the performance classes to polynomial functions.

Once the fuzzer reports a partition of the input space into sets with distinguishable performance classes, the debugging problem is to find an explanation of the distinction between various performance classes. Oftentimes, this explanation can not be reported based solely on the values of the input variables and is a function of syntactic structure ( $Y$  variables) of the program. To enable such explanations, we study the discriminant learning problem, where the goal is to learn the differences between various partitions of the inputs as predicates over the trace variables.

A predicate over the execution space  $\mathcal{D}_Y$  is a function  $\phi : \mathcal{D}_Y \rightarrow \{\top, \perp\}$ . Let  $\Phi(\mathcal{D}_Y)$  be the set of predicates over  $\mathcal{D}_Y$ . Given an input partition  $P = \{D_1, D_2, \dots, D_k\}$  of a program  $\llbracket \mathcal{P} \rrbracket = (X, Y, O, \pi)$ , a discriminant is function  $\Delta : P \rightarrow \Phi$  such that for every  $D_i \in P$  we have that  $x \in D_i$  implies  $\Delta(D_i)(O(x)) = \top$ .

**DEFINITION 3.3 (DIFFERENTIAL PERFORMANCE DEBUGGING).** Given a program  $\mathcal{P} = (X, Y, O, \pi)$  and an input partition  $P = \{D_1, \dots, D_k\}$ , the differential performance debugging problem is to find a discriminant  $\Delta : P \rightarrow \{\top, \perp\}$  of  $P$ .

Tizpaz-Niari et al. [39] showed that the differential performance debugging problem is already NP-hard for programs with a finite set of inputs. In Section 4.2, we present a data-driven approach to learn discriminants as decision trees.

### 4 DATA-DRIVEN APPROACH

To address problems 3.2 and 3.3, we propose a data-driven approach in two steps. First, we extend gray-box evolutionary fuzzing algorithms [1, 9] to generate performance differentiating inputs. Our fuzzer is equipped with clustering to identify performance classes while generating inputs. Second, we use discriminant learning [38] to pinpoint code regions connected to the differential performance.

#### 4.1 Differential Performance Fuzzing

The overall fuzzing algorithm is shown in Algorithm 1. Given a program  $\mathcal{P}$ , the goal is to discover widely varying performance classes. Next, we describe important components of our evolutionary search based Algorithm 1.

**Cost Measure.** The performance (cost) model  $\pi$  summarizes the resource usages. We consider both 1) abstractions of resource usages such as the number of lines executed and 2) concrete resource usages such as the execution times.

**Trace Summary.** We consider an instantiation of trace summary function  $O$  for fuzzing where the function  $O$  takes an input  $x \in X$  and returns a set of edges in the control flow graph (CFG) visited during the execution of the input  $x$ .

**Path Model.** We characterize the subset  $S$  of input space using information from program traces. In particular, we use the path information to determine if two inputs are in the same class  $s \in S$ . We represent paths using the hash values of their ids. For each edge in the CFG, we consider a unique edge id. Then, we apply a hash function  $H$  on the set of edge ids visited for executing an input. Two distinct inputs  $x_1, x_2 \in X$  are in the same performance class if  $H(y_1) = H(y_2)$  where  $y_1 = O(x_1)$  and  $y_2 = O(x_2)$ .

**Algorithm 1:** DPFUZZ: EVOLUTIONARY FUZZING FOR DIFFERENTIAL PERFORMANCE.

---

**Input:** Program  $\mathcal{P}$ , initial seed  $X$ , max. number of iterations  $N$ , steps to do clustering  $M$ , the tolerance  $\epsilon$ , the separation  $\varsigma$ , num. of clusters  $k$ .

**Output:** Inputs/clusters manifest performance of  $\mathcal{P}$ .

```

1 for  $x \in X$  do
2    $path, cost \leftarrow run(\mathcal{P}, x)$ 
3    $Cov[path].add(cost), Pop[path].add(s)$ 
4    $clusters \leftarrow clust(Cov, Pop, \epsilon)$ 
5    $step \leftarrow 1$ 
6   while  $step \leq N \wedge |clusters| < k$  do
7     if  $step \% M = 0$  then
8        $clusters \leftarrow clust(Cov, Pop, \epsilon)$ 
9        $cluster \leftarrow choice_R(clusters)$ 
10       $path \leftarrow choice_W(cluster)$ 
11       $x \leftarrow choice_W(Pop[path])$ 
12       $x' \leftarrow mutate(x)$ 
13       $path', cost' \leftarrow run(\mathcal{P}, x')$ 
14      if  $path' \notin Cov.keys()$  then
15         $Cov[path'].add(cost'), Pop[path'].add(inp')$ 
16      else if promisingInput( $cost', path', n$ ) then
17         $Cov[path'].add(cost'), Pop[path'].add(inp')$ 
18       $step \leftarrow step + 1$ 
19   return  $Cov, Pop, clusters$ .

```

---

**New Input.** We add an input to the population if the path is new, i.e. the id of the path is not in the coverage key (line 14 in Algorithm 1), or the input has a higher cost in comparison to other inputs in the same path (line 16 in Algorithm 1).

**Functional Data Clustering.** We use non-parametric functional data clustering algorithms [6] to cluster paths into a few similar performance groups (line 4 and 8 in Algorithm 1). For a set of inputs  $x_1, \dots, x_n$  mapped to the same path  $h$ , i.e.  $H(y_1) = \dots = H(y_n) = h$ , we fit linear  $a|x| + b$  and polynomial  $a|x|^b$  functions [4] to model the performance function  $\pi_h$ . Then, we calculate the  $l_1$  distances between the performance functions and apply KMeans clustering [10] with the tolerance bound  $\epsilon > 0$  on the distance matrix to partition the paths into  $k$  clusters  $\Sigma$ . The clustering guarantees the following condition: if two paths (and their corresponding performance functions) are in the same cluster ( $h, h' \in \Sigma_i$ ), then  $d_1(\pi_h, \pi_{h'}) \leq \epsilon$ .

The  $clust$  function in Algorithm 1 works as the following: starting with one cluster ( $k = 1$ ), we apply KMeans clustering and check if the condition holds, i.e. all the functions in the same cluster are  $\epsilon$  close to each other. If this is the case, we return the clusters. Otherwise, we increase  $k$  to  $k + 1$  and run the algorithm again. The clustering algorithm helps explore (few) paths with distinguishable performance functions as opposed to (too many) paths in the program. Each cluster contains one or more paths that have similar performance behaviors. The selection function (line 10 in Algorithm 1) chooses a path inside a cluster based on weighted probabilities where a path with higher score of cost has a better chance of selection. Similar criterion has used to choose an input

from the set of possible inputs inside the path population (line 11 in Algorithm 1).

**Mutations and Crossover.** We consider 8 well-established [1, 8, 16] mutation operations to guide the search algorithm. We also consider crossover operation where it mixes the current input with another input from the population.

**Termination Conditions.** The Algorithm 1 terminates either if it reaches to the maximum steps  $N$  or there are  $k$  clusters with significant performance differences (line 6 in Algorithm 1).

## 4.2 Differential Performance Debugging

Given the set of inputs  $Pop = \{X_1, \dots, X_k\}$  and their performance label  $\Sigma$  from the fuzzing step, the algorithm 2 explains the differential performance in the inputs. First, the user of DPFuzz (optionally) runs the clustering algorithm with the parameter  $k$  to obtain performance clusters (line 1 in Algorithm 2). The debugging procedure uses the inputs  $Pop$  as features and their clusters as labels to learn a decision tree model that explains the differential performance in the space of input parameters. We use CART decision tree algorithms [2] to learn the set of predicates  $P$  in the space of input parameters (line 3 in Algorithm 2). For example, in the decision tree of Figure 1, solver='saga'  $\wedge$  multi-class='multinomial'  $\wedge$  penalty='l2' is the predicate for the green cluster such that inputs satisfying these parameters belong to the green cluster.

The critical step in debugging is to explain the differences based on the program internals. For this step, the inputs  $Pop$  are feed into the instrumented program  $\mathcal{P}'$  that generates program internal features such as whether a method or a condition are invoked and how many times they are called (line 4 in Algorithm 2). Given the set of (internal) features  $\{\mathcal{P}'(X_1), \dots, \mathcal{P}'(X_k)\}$  and their cluster labels  $\Sigma$ , the problem of discriminant learning becomes a standard classification problem. We use CART algorithms to learn the set of predicates  $\Phi$  in the space of program internal features (line 5 in Algorithm 2). These predicates partition the space of internal features into hyper-rectangular sub-spaces  $\{\phi_1, \dots, \phi_k\}$  such that if  $H(O(x)) \in \Sigma_j$ , then  $\mathcal{P}'(x)|=\phi_j$ , i.e., the predicate  $\phi_j$  evaluates to true for the evaluation of input  $x$  trace feature given that the input  $x$  is in the performance cluster  $j$ . For example, a predicate that evaluates whether a particular method is invoked determines the complexity of its performance class. In Figure 1 (d), the predicate based on whether the number of calls to the condition `np.max(absgrad)` is more than 14 distinguishes the blue and black clusters. The debugger uses this information to localize regions in the code and to potentially fix a performance bug.

## 5 EXPERIMENTS

### 5.1 Implementation Details

**Environment Setup.** We use a super-computing machine for running our fuzzer. The machine has a Linux Red Hat 7 OS with 24 cores of 2.5 GHz CPU each with 4.8 GB RAM. Since the performance measure for machine learning libraries is the actual execution times (noisy observations), we re-run the generated inputs from the fuzzer on a more precise but less powerful NUC5i5RYH machine and use the measurements of this machine for clustering. We consider the version 2.7 of python and 0.20.3 of scikit-learn.

**Algorithm 2:** DPFUZZ: EXPLAINING DIFFERENTIAL PERFORMANCES.

---

**Input:** Program  $\mathcal{P}$ , instrumented program  $\mathcal{P}'$ , desired clusters  $k$ , inputs  $Pop$ , the class *label*.  
**Output:** The set of predicates  $P, \Phi$  explain differential performances.

```

1 label  $\leftarrow$  clust(Pop, k)
2  $P \leftarrow$  DecisionTree(Pop, label)
3  $Y \leftarrow \mathcal{P}'(\textit{Pop})$ 
4  $\Phi \leftarrow$  DecisionTree(Y, label)
5 return  $P, \Phi$ .
```

---

**Fuzzing and Clustering.** We implement the fuzzing of DPFuzz in python by extending the Fuzzing Book framework [43]. The implementation is over 900 lines of code and can fuzz both python and Java applications. The performance measure is the actual running times in case-studies and the number of executed lines in micro-benchmarks. We use trace library [14] (python) and Javassist [3] (Java) to model paths and measure performances. We use numpy polynomial module [12] to fit performance functions. We implement the KMeans clustering algorithm using scikit-learn [15].

**Debugging and Instrumentation.** We instrument python libraries with tracing [14] and Java applications with Javassist [3] to extract internal features. We implement the decision tree classifier using CART algorithm in scikit-learn [15].

## 5.2 Micro-benchmark Results

We compare our fuzzing technique DPFuzz against state-of-the-art performance fuzzers. For the benchmark, we consider standard sorting, searching, tree, and graph algorithms from [35]. These benchmarks are standard programs used to evaluate performance fuzzers. We consider SlowFuzz [16] and PerfFuzz [8] from the literature. All three fuzzers share the same functionality such as mutations. The differences are in the population model, adding a new input to the population, and choosing an input from the population.

**SlowFuzz.** The SlowFuzz [16] aims to find the worst-case algorithmic complexity. The fuzzing approach has a global population where it adds a new input to the population if it achieves a higher cost (performance measure) than any other inputs in the population. The fuzzing approach chooses an input for mutations from the current population randomly.

**PerfFuzz.** This fuzzing [8] aims to find the worst-case algorithmic complexity for each entity (such as an edge) in the CFG. The fuzzing has a global population, and it adds a new input to the population if the input has visited a new edge (discovered a new path) or the input has achieved the highest cost in visiting at least one edge. It picks an input for mutation based on whether the input has had the highest cost for at least one entity.

**DPFuzz.** The fuzzing follows Algorithm 1 where there are multiple populations, one for each unique path in the CFG. An input is added to the population if the path induced from it has visited a new edge in the CFG (forms a new population) or the input has the highest cost in the population of this path. DPFuzz performs clustering after many steps (set to 1,000 in experiments) and uses the clustering information to pick an input from the population.

**Empirical Research Question.** For a given program over a fixed time of fuzzing, we compare the fuzzing techniques on 5 criteria: the number of generated inputs, the worst-case computational complexity in terms of executed lines, the number of visited unique paths, the number of unique performance functions, and the number of clusters of performance functions. We repeat each experiment for different fuzzers 5 times and report the best results obtained by the fuzzers. Table 1 shows the outcome of each fuzzing technique on 5 different criteria for 10 different algorithms.

**Number of generated samples.** We fix the duration of fuzzing to be 90 minutes for different fuzzers in all benchmarks. We compare the number of inputs generated with different fuzzers. Note that the quality of inputs such as the ones with higher costs of executions affects the number of generated inputs. In general, SlowFuzz and PerfFuzz could generate inputs faster in comparison to DPFuzz. Examples such as Quick sort, Merge sort, and Binary search provide fair comparison in terms of generated inputs since all fuzzers have similar performances in finding worst-case execution times. The slowdown in DPFuzz is almost 2x compared to SlowFuzz and PerfFuzz. We emphasize that the slowdown is expected due to functional fitting and clustering in DPFuzz.

**Worst-case cost of execution.** We examine the fuzzers outcomes in finding inputs with the highest cost in terms of executed lines. Table 1 shows that DPFuzz finds inputs with higher costs in 5 out of 10 benchmarks in comparison to SlowFuzz and PerfFuzz.

**Discovered paths.** We consider the number of unique paths discovered by the fuzzers. A path is unique if it has visited an edge that is not visited by any other paths. Table 1 shows SlowFuzz has discovered the fewest paths. In 3 out of 10 cases, PerfFuzz has discovered more paths compared to other fuzzers.

**Number of distinct performance functions.** One requirement of characterizing performance classes is to have multiple inputs (varied by size) for a path and fit performance functions. We compare the number of performance functions discovered by the fuzzers. Table 1 shows DPFuzz finds more performance functions in 7 out of 10 benchmarks.

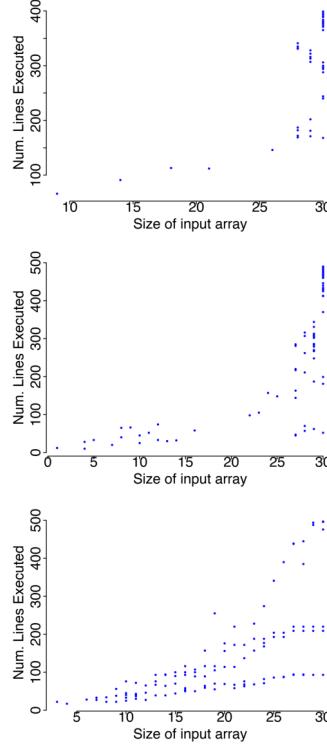
**Number of functional clusters.** We consider the number of clusters in performance functions to show performance classes. We use the  $l_1$  distance between functions for clustering. We note that the number of clusters should be chosen based on the quality of clustering for fair comparisons. As we increase the number of clusters, we measure the sum of intra-cluster distances as the error. We pick the optimal number of clusters when the error is below 1,000 in accordance with standard elbow method for choosing ideal number of clusters. Since the error value is the same in all experiments, the resulting number of clusters is a fair indication of detecting differential performance. Table 1 shows DPFuzz finds more clusters in 4 out of 10 benchmarks.

**Summary.** Although DPFuzz is slower in generating inputs, it outperforms other fuzzers in finding worst-case costs, performance functions, and clusters. Figure 3 shows an example of Optimized Insertion sort (InsertionX). The plots are the performances (in terms of executed lines) versus the size of inputs for  $2 * 10^6$  samples generated by the fuzzers.

**Table 1: Micro-benchmark results. Comparing DPFuzz vs SlowFuzz [16] and PerfFuzz [8]. Legend: #L: lines of code, #N: number of generated samples, T: fuzzing time (min), W: worst-case computation cost (in term of executed lines), #P: number of unique paths, #M: Number of distinct performance functions, #K: Number of functional clusters. Note:  $M = 10^6$  and  $K = 10^3$ .**

Algorithm	#L	T	SlowFuzz [16]					PerfFuzz [8]					DPFuzz				
			#N	W	#P	#M	#K	#N	W	#P	#M	#K	#N	W	#P	#M	#K
Quick Sort	80	90	13.6M	721	5	4	2	10.8M	716	14	8	3	7.8M	721	14	11	3
3-Ways Q-Sort	83	90	12.0M	756	4	2	1	12.1M	801	10	6	3	7.1M	847	10	8	5
InsertionX Sort	42	90	15.0M	496	3	2	1	15.2M	490	10	4	2	10.9M	497	10	9	4
Merge Sort	53	90	24.9M	516	4	1	1	22.5M	516	6	5	1	10.4M	516	6	5	1
Binary Search	75	90	11.2M	530	7	4	1	11.2M	527	35	21	6	5.7M	529	34	26	6
Seq. Search	26	90	43.8M	60	2	2	1	50.0M	60	6	3	1	13.2M	72	6	4	1
Boyer Moore	88	90	29.2M	204	4	0	0	34.7M	372	8	1	1	9.6M	372	8	1	1
BST Insert	47	90	21.2M	501	6	3	1	18.1M	561	13	12	3	7.7M	566	13	12	5
Is BST	141	90	22.3M	280	14	7	1	26.3M	224	46	19	2	10.0M	280	40	25	2
Prim's MST	294	90	7.9M	1,006	10	5	2	7.9M	975	119	23	5	5.3M	1,020	118	70	11

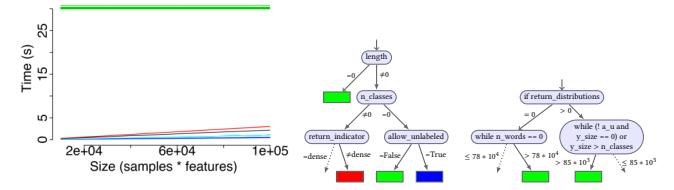
```
Listing 1: InsertionX
def insertionX(a,n):
    exchange, i = 0, n-1
    while i > 0:
        if a[i] < a[i-1]:
            swap(arr,i,i-1)
            exchange += 1
        i -= 1
    if exchange == 0:
        return a
    i = 2
    while i < n:
        v, j = a[i], i
        while v < a[j-1]:
            a[j] = a[j-1]
            j -= 1
        a[j], i = v, i+1
    return a
```



**Figure 3: InsertionX. Results for SlowFuzz [16], PerfFuzz [8], and DPFuzz, from top to bottom, respectively.**

## 6 ML LIBRARY ANALYSIS

We analyze 8 larger machine learning (ML) libraries from scikit-learn [15]. Although our approach is general enough to apply to any software library and system (we currently support both Python and Java applications), there are properties in ML applications that make our approach more practical. In particular, there is usually a clear distinction between the parameters of the library and the



**Figure 4: (a) Inputs of make classification are clustered into 5 groups. (b) Decision tree model based on input parameter features. (c) Decision tree model based on internal features.**

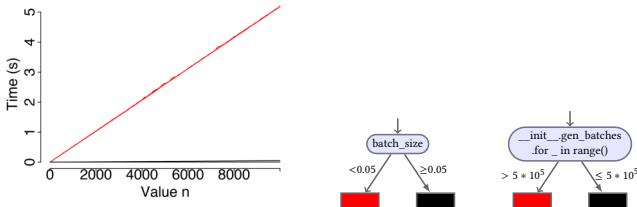
data in ML libraries. Once we fix the parameters in the given ML algorithm, the execution times are often a simpler function (e.g. linear and quadratic) of the number of samples and features in the data. General programs often do not follow such a structured discipline. The main research questions are “does DPFuzz (a) scale well for real-world ML libraries and (b) provide useful information to debug performance issues?”

**A) Logistic Regression Classifier.** In summary, DPFuzz detects 4 clusters after fuzzing for about 2 hours. The debugging revealed a performance bug in the implementation of logistic regression that has since been fixed (see details in Overview section 2).

**B) Make Classification Data Set Util.** We analyze the performance of `make_multilabel_classification` method inside `sample_generator` module [25].

*Fuzzing and clustering.* DPFuzz provides 243 sets of inputs related to different paths in the module after running for 4 hours. The fuzzing covers 293 LoC from almost 350 LoC in the implementations of this method and its dependencies. Figure 4 (a) shows that the 243 performance functions are clustered into 5 groups. The clustering shows a huge differential performance between the green cluster and other clusters.

*Analyzing input space.* Figure 4 (b) shows that if the length parameter sets to 0, then the input is in the green cluster. Another way to see the expensive green cluster is to set the value of `n_class` parameter to 0 and `allow_unlabeled` parameter to False.



**Figure 5:** (a) Inputs generated by DPFuzz for util module clustered. (b) Decision tree using the input parameters of util. (c) Decision tree using the internal features of util.

*Bug localization.* The decision tree model in Figure 4 (c) shows the green cluster is associated with the calls to two different loops. The following code snippet shows these parts in `make_multilabel_classification` module:

```
def sample_example():
    ...
    while (not allow_unlabeled and
          y_size == 0) or y_size > n_classes:
        y_size = generator.poisson(n_labels)
    ...
    while n_words == 0:
        n_words = generator.poisson(length)
    ...

    ...
    weights = [1.0 / n_classes] * n_classes
    ...
    weights[k % n_c] / n_clusters_per_class
```

The code snippet shows the possibility of an infinite number of executions for the two loop bodies (the fuzzer terminates processes if the execution of inputs takes more than 15 minutes). We mark these two parts in `make_multilabel_classification` method performance bugs ②, ③. We have reported these issues to scikit-learn developers [30]. They confirmed the bugs and have since fixed them [31]. The fix does not allow parameters `n_classes` and `length` to accept zero values. There are also two division by zero crashes in `make_classification` method discovered during fuzzing if the `n_classes` parameter or `n_clusters_per_class` parameter set to zero:

```
...
weights = [1.0 / n_classes] * n_classes
...
weights[k % n_c] / n_clusters_per_class
```

*Scalability.* In 240 minutes, DPFuzz generates 243 performance functions. For debugging, DPFuzz generates 293 internal features and uses the features to learn the decision tree in 1(s).

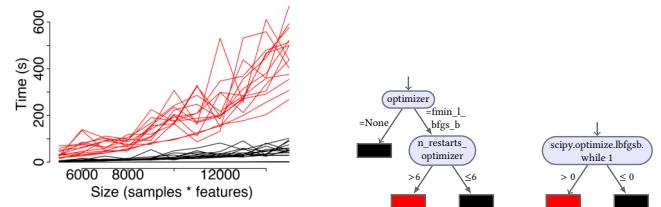
*Usefulness.* DPFuzz discovers and pinpoints 2 performance bugs and 2 division by zero crashes in the implementations of `make_multilabel_classification` method.

**C) Batch Generator.** We analyze the implementation of util<sup>3</sup> in [15]. This module provides various utilities such as generating slices of certain sizes for the given data.

*Fuzzing and clustering.* We fuzz batch generator methods of this module for 30 minutes and obtain 20 sets of inputs. Figure 5 (a) shows that there are two clusters of performance.

*Analyzing input space.* Figure 5 (b) shows that the differences between red and black patterns are related to `batch_size` parameter and the library accepts small positive float values for this parameter such as 0.001.

<sup>3</sup>[https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/\\_\\_init\\_\\_.py](https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/__init__.py)



**Figure 6:** (a) Inputs generated by DPFuzz for Gaussian classifier clustered. (b) Decision tree using the input parameters of Gaussian. (c) Decision tree using the internal features of Gaussian.

*Bug localization.* Figure 5 (c) pinpoints the following loop body in `gen_batches()`:

```
for _ in range(int(n // batch_size)):
    end = start + batch_size
    ...

```

The decision tree shows that the loop above can be taken millions of times if the `batch_size` sets to small positive values close to 0. We mark this as performance bug ④. This bug has since been confirmed and fixed by the developers [33, 34]. The fix checks the `batch_size` parameter to be both integer and greater than or equal to 1.

*Scalability.* During 30 mins of fuzzing, DPFuzz generates 20 performance functions. DPFuzz generates 15 internal features and uses the features to infer the decision tree in 0.1(s).

*Usefulness.* DPFuzz discovers and pinpoints a performance bug in the util module.

**D) Gaussian Process Classification.** We analyze the implementations of Gaussian Process (GP) as a classifier model [22] in the scikit-learn library [15]. This classifier is specifically used for probabilistic classification (see [21] for more details about the functionality of this classifier). We fix non-deterministic (stochastic) behaviors in the library to a deterministic random value.

*Fuzzing and clustering.* We run DPFuzz for 240 minutes and obtain 173 sets of inputs. Figure 6 (a) shows a huge performance difference between the black and red clusters (more than 10 times).

*Analyzing input space.* Figure 6 (b) shows that if the optimizer parameter sets to '`fmin_l_bfgs_b`' and `n_restarts_optimizer` parameter sets to values more than 6, then the input is in the (slow) red cluster. Otherwise, the input follows the (fast) black cluster.

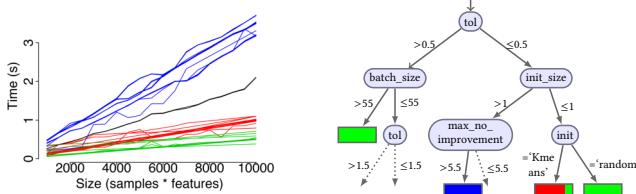
*Bug localization.* Figure 6 (c) shows that the number of calls to the loop body inside `scipy.optimize.lbfgsb` module causes the performance differences:

```
while 1:
    ...
    _lbfgsb.setulb(...)
    ...

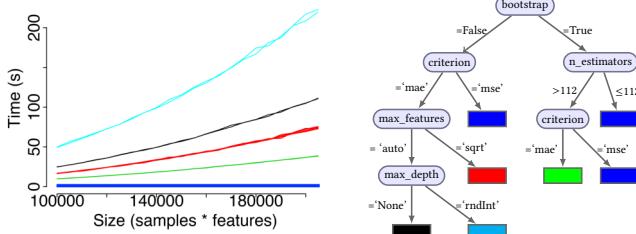
    ...

```

It seems that the Gaussian classifier runs for the number of `n_restarts_optimizer` parameter and it causes huge performance differences by calling to `lbfgsb` optimizer. Since the behavior is related to external library, we left further analysis for future work to determine if the behavior is intrinsic to the problem, or it is a performance bug.



**Figure 7: (a) Inputs generated by DPFuzz for mini-batch KMeans are clustered into 4 groups. (b) Decision tree using the input parameters of KMeans.**



**Figure 8: (a) Inputs generated by DPFuzz for forest regressor are clustered into 5 groups. (b) Decision tree using the input features of forest regressor.**

**Scalability.** During 240 mins, DPFuzz obtains 173 performance functions. DPFuzz generates 1,333 features about program internals and uses the features to learn the decision tree model in 5(s).

**Usefulness.** DPFuzz discovers and pin-points a code region in an external library (scipy optimizer).

**E) Mini-batch KMeans.** We analyze the implementations of KMeans mini-batch clustering [26]. This is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimize the same objective function. We assume that the initial cluster centroids are deterministically chosen by setting seed values to a constant value.

During a run of 240 mins, we obtain 325 input sets from DPFuzz. Figure 7 (a) shows that 325 functions are clustered in 4 groups. Figure 7 (b) shows the expensive blue cluster happens if the ‘tol’ is less than or equal to 0.5, the ‘init\_size’ is larger than or equal to 1, and the ‘max\_no\_improvement’ is larger than 5.5. Looking into the internal features, we observe that the number of calls to calculate the euclidean distance between points and the squared differences between the current and the previous errors are important discriminant features. However, the discriminant features seem to explain behaviors that are intrinsic to the problem.

**F) Random Forest Regressor.** A random forest [28] is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. We analyze its implementations in scikit-learn library [15].

We obtain 160 sets of inputs from DPFuzz after running for 240 mins. Figure 8 (a) shows (subset of) inputs are clustered into 5 groups. The clustering shows that the computational complexity

in the random forest regressor can be quadratic. The decision tree model in Figure 8 (b) shows that the inputs with criterion parameter sets to ‘mae’ have higher costs. A similar issue has already reported as a potential performance bug in the regressor [18]. Learning forest regressors is expensive with ‘mae’ criteria since it sorts inputs in each step of learning to calculate the median. Since the root causes seem to be related to external library, we left further analysis for future work to determine if the behavior is intrinsic to the problem, or it is a performance bug.

**G) Discriminant Analysis.** The discriminant analysis [23] is a classic classifier with linear and quadratic decision boundaries. We analyze linear and quadratic discriminant analysis implemented in scikit-learn [15]. During 52 mins of fuzzing, we obtain 78 sets of inputs. Upon clustering, we observe that the point-wise distances between clusters are in order of fractions of a second. Therefore, we gain more confidence that the discriminant analysis is free of performance issues.

**H) Decision Tree Classifier.** The decision tree classifier [20] is a white-box classification model that predicts the target variable by inferring decision rules. We analyze the implementations of this model in scikit-learn [15]. After fuzzing for 240 mins, DPFuzz generates 492 sets of inputs. Upon clustering, we realize the point-wise distances between centroids are less than 0.1 second. Thus, we become more confident that the decision tree classifier is free of performance bugs.

## 7 RELATED WORK

**Performance Fuzzing.** Evolutionary algorithms have been widely used for finding inputs that trigger the worst-case complexity [8, 16]. SlowFuzz [16] extends the libFuzzer [9] to discover DoS bugs, that is, inputs with expensive computations such as exponential. In contrast, DPFuzz is looking for different classes of computational complexities rather than only the worst-case one. PerfFuzz [8] is the closest fuzzing technique to us. The goal is to maximize the cost of different entities in the program (edges in CFG) that help characterize the worst-case behaviors in large-scale systems. PerfFuzz has a single global population whereas DPFuzz has multiple populations, one population per path. This model of population in DPFuzz enables the debugger to model performance functions precisely and obtain diverse classes of performances. In addition, DPFuzz utilizes clustering during fuzzing to discover a few diverse performance classes rather than all entity classes, many of those may have similar performance.

**Differential Fuzzing.** DiffFuzz [11] has developed on top of AFL [1] and Kelinci [7] to discover information leaks due to timing side channels in Java programs. DiffFuzz adapts the traditional notion of confidentiality, noninterference. A program is *unsafe* iff for a pair of secret values  $s_1$  and  $s_2$ , there exists a public value  $p$  such that the behavior of the program on  $(s_1, p)$  is observably different than on  $(s_2, p)$ . The goal of DiffFuzz is to maximize the following objective:  $\delta = |c(p, s_1) - c(p, s_2)|$ , that is, to find two distinct secret values  $s_1, s_2$  and a public value  $p$  that give the maximum cost ( $c$ ) difference in two runs of a program. If the difference of any pair of secret values is more than  $\epsilon$ , the program is considered to be vulnerable to timing side-channel attacks. In contrast to DiffFuzz, we are looking

for  $k$  sets of inputs to discover differential performances in python-based machine learning libraries. Fuchsia [40] is another technique that performs differential analysis to debug timing side channels. For detection, Fuchsia extends AFL to characterize response times as functions over public inputs. Fuchsia performs fuzzing and clustering in two separate steps, whereas DPFuzz combines these two steps to explore the space of input efficiently. In addition, the time model in Fuchsia can be in arbitrary shapes, while the performance model in DPFuzz often follows simpler shapes such as polynomials. **Debugging Performances.** Machine learning and statistical models have been used for fault localization [42] and debugging of performance issues [36, 38, 39]. These works generally assume that the interesting inputs are given, while we adapt evolutionary-based fuzzing techniques to automatically generate interesting inputs. DPDEBUGGER [38] considers a model of programs where the inputs are not gathered as functional data. Therefore, it needs to discover performance functions. DPDEBUGGER extends Kmeans and Spectral clusterings to find clusters of performance from independent data points. On the contrary, DPFuzz considers a model of programs where inputs are given as functional data. Then, it uses non-parametric functional data clustering [6] to detect clusters of performance. DPDEBUGGER [38] is limited to linear performance functions, while DPFuzz can model complex functions such as polynomials. Similar to [38], DPFuzz uses decision tree classifiers to pinpoint code regions contributing to the differential performance.

## 8 THREAT TO VALIDITY

**Evolutionary-based Fuzzing.** Our approach requires a diverse set of inputs generated automatically using the fuzzing component. The quality of the debugging significantly depends on the characterization of differential performance in the given input set. Similar to existing evolutionary-based fuzzers, our approach relies solely on heuristics to generate a diverse set of inputs and is not guaranteed to find inputs that characterize all performance classes.

**Benign Differential Performance vs Performance Bugs.** In general, it is indeed challenging to determine whether a differential performance is a bug or it is intrinsic to the problem being solved. To mitigate this issue, we turn to debugging with the help of auxiliary features from the space of inputs and internals.

First, we find an explanation based on the input features such as the type of solver. Based on this, the user may decide the differences due to using solver='A' versus solver='B' are benign whereas the differences due to using tolerance=0.0 versus tolerance=0.000001 under the same solver='A' are unexpected.

Once an unexpected behavior is detected, the next step is to investigate the issue further in the source code and localize suspicious code regions for a fix. These two steps help the user determine whether the differences are intrinsic, or they are performance bugs that need to be fixed.

**Experiments and Comparisons to Existing Fuzzers.** Existing approaches in fuzzing such as SlowFuzz [16] and PerfFuzz [8] are mainly developed for C and C++ programs and extended recently for Java applications [11]. Our work provides substantial (and unprecedented) support specifically for python-based ML libraries.

To enable ourselves to compare DPFuzz against the existing performance fuzzers, we adapt their main fuzzing algorithm and

implement them in our python-based fuzzing framework. This, however, can lead to degradations in the performance of these fuzzers. To alleviate this in our comparisons, we use the same functionality in all aspects of three fuzzers such as mutation and crossover operations. The differences are in modeling population (multiple populations with clustering versus single global population) and adding new inputs to the population (based on the cost, the path, or combinations). These differences are the specific choices of each fuzzer to achieve certain goals as described in their algorithms.

**Overhead in Dynamic Analysis.** We proposed a dynamic analysis approach to discover and understand performance bugs. Dynamic analysis often scales well to large applications. However, as compared with static analysis, they present additional overheads such as time required to discover variegated inputs and time needed for data collection.

**Polynomial Functions and Decision Tree Models.** Our design guiding principles are based on two important factors: efficiency (especially for fuzzing) and human interpretability (for debugging). For example, we restrict the search for performance functions to be polynomials so as to generate inputs quickly. Other models such as Gaussian processes can lead to better results, but they may degrade the throughput of fuzzing. Similarly, we use decision tree models to give interpretable explanations. Graph models can be used to learn complex discriminants and overcome the decision tree limitations such as the hyper-rectangular partitions of search spaces. However, such models are notorious to be uninterpretable.

**Standard Algorithms as Benchmarks.** We use standard sort, search, tree, and graph algorithms to evaluate our fuzzing. While these algorithms do not contain performance bugs, they have (well-known) diverse classes of performances, and finding those classes is a hard problem. Characterizing these classes in the well-known algorithm through fuzzing is important to manifest differential performance bugs in real-world applications.

**Machine Learning Libraries as Case Studies.** In this work, we focus on medium-sized ML libraries for few reasons. First, there is a little bit of support for the performance aspects of ML libraries. Second, there is often a clear distinction between data and parameters in ML libraries, and they tend to have simpler performance functions such as linear or polynomial. Our approach is applicable for general software given that there is a clear measure defined to map inputs to the size. Examples are the number of bytes in a file, the number of set bits in a key, and the number of Kleene star in a regular expression. Given this measure, our approach can characterize performance differences and aid to localize the root causes. We left further analysis to apply this technique on general and large software for future work.

**Time Measurements.** We use actual execution times as opposed to abstractions such as the number of executed lines in the case studies. While this is important to factor the cost of black-box components (such as external libraries and solvers in other languages) during the fuzzing, the noise in timing observations can lead to false positive in the fuzzing process. To overcome this issue, we re-run the inputs once more on NUC5i5RYH machine to allow for higher precisions. To further mitigate the effects of environmental factors in NUC measurements, we run the libraries in isolations and take the average of timing measurements over multiple samples.

## 9 CONCLUSION AND FUTURE WORK

We developed a method and a tool for differential performance analysis. We showed that the fuzzing, clustering, and decision tree algorithms presented for functional data are scalable to debug real-world machine learning libraries. In addition, we illustrated the usefulness of our approach in finding multiple performance bugs in these libraries and in comparing to existing performance fuzzers.

For future work, there are few interesting directions. One direction is to study security implications of differential performance. The feasibility of (hyper)parameter leaks [41] via timing side channels in ML applications is a relevant and challenging open problem. Another direction is to study the relationships between accuracy and performance. Given a lower-bound on the accuracy of a learning task, the idea is to synthesize parameters and hyper-parameters in the model such that the performance of underlying systems such as IoT and CPS are optimized.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments to improve this paper. In addition, the authors thank the developers of `scikit-learn` for discussing the potential issues reported by us. This work utilized resources from the University of Colorado Boulder Research Computing Group, which is supported by NSF, CU Boulder, and CSU. This research was supported by DARPA under agreement FA8750-15-2-0096.

## REFERENCES

- [1] AFL. 2016. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>. Online.
- [2] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees*. CRC press.
- [3] Shigeru Chiba. 1998. Javassist - a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vol. 174.
- [4] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *FSE*. ACM, 395–404.
- [5] J. Hartmanis and R. E. Stearns. 1965. On the Computational Complexity of Algorithms. *Trans. Amer. Math. Soc.* 117 (1965), 285–306. <http://www.jstor.org/stable/1994208>
- [6] Julien Jacques and Cristian Preda. 2014. Functional data clustering: a survey. *Advances in Data Analysis and Classification* 8, 3 (2014), 231–255.
- [7] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *CCS*. ACM, 2511–2513.
- [8] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perfuzz: Automatically generating pathological inputs. In *ISSTA*. ACM, 254–265.
- [9] libFuzzer. 2016. A library for coverage-guided fuzz testing (part of LLVM 3.9). <http://llvm.org/docs/LibFuzzer.html>. Online.
- [10] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.
- [11] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: Differential Fuzzing for Side-Channel Analysis. *ICSE* (2019). <http://arxiv.org/abs/1811.07005>
- [12] Travis Oliphant. 2006–. NumPy: A guide to NumPy. <http://www.numpy.org/>.
- [13] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, Vol. 50. ACM, 369–378.
- [14] Zooko O'Whielacronx. 2018. A program/module to trace Python program or function execution. <https://github.com/python/cpython/blob/2.7/Lib/trace.py>. Online.
- [15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [16] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS*. ACM, 2155–2168.
- [17] James O Ramsay. 2006. *Functional data analysis*. Wiley Online Library.
- [18] Scikit-learn. 2017. Trees with MAE criterion are slow to train. <https://github.com/scikit-learn/scikit-learn/issues/9626>. Online.
- [19] Scikit-learn. 2018. Sqeuclidean metric is much slower than euclidean. <https://github.com/scikit-learn/scikit-learn/issues/12600>. Online.
- [20] Scikit-learn. 2019. Decision Tree Classifier. <https://scikit-learn.org/stable/modules/tree.html>. Online.
- [21] Scikit-learn. 2019. Gaussian Process Classifier in scikit-learn: description. [https://scikit-learn.org/stable/modules/gaussian\\_process.html#gaussian-process-classification-gpc](https://scikit-learn.org/stable/modules/gaussian_process.html#gaussian-process-classification-gpc). Online.
- [22] Scikit-learn. 2019. Gaussian Process Classifier in scikit-learn: implementations. [https://scikit-learn.org/stable/modules/generated/sklearn.gaussian\\_process.GaussianProcessClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessClassifier.html). Online.
- [23] Scikit-learn. 2019. Linear and quadratic discriminant analysis. [https://scikit-learn.org/stable/modules/lda\\_qda.html](https://scikit-learn.org/stable/modules/lda_qda.html). Online.
- [24] Scikit-learn. 2019. Logistic Regression in scikit-learn. [https://scikit-learn.org/stable/modules/linear\\_model.html#logistic-regression](https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression). Online.
- [25] Scikit-learn. 2019. Make Multilabel Classification. [https://scikit-learn.datasets.make\\_multilabel\\_classification](https://scikit-learn.datasets.make_multilabel_classification).
- [26] Scikit-learn. 2019. Mini-batch KMeans. <https://scikit-learn.cluster.MiniBatchKMeans>.
- [27] Scikit-learn. 2019. Performance of Logistic Regression with saga. <https://github.com/scikit-learn/scikit-learn/issues/13316>. Online.
- [28] Scikit-learn. 2019. Random Forest Regressor. <https://scikit-learn.ensemble.RandomForestRegressor>.
- [29] Scikit-learn. 2020. Performance bug in logistic regression with newton-cg. <https://github.com/scikit-learn/scikit-learn/issues/16186>. Online.
- [30] Scikit-learn. 2020. Performance bug in Make Classification Data Set Util. <https://github.com/scikit-learn/scikit-learn/issues/16001>. Online.
- [31] Scikit-learn. 2020. Performance bug in Make Classification Data Set Util fixed. <https://github.com/scikit-learn/scikit-learn/pull/16006/files>. Online.
- [32] Scikit-learn. 2020. Performance bug in regression with newton-cg fixed. <https://github.com/scikit-learn/scikit-learn/pull/16266/files>. Online.
- [33] Scikit-learn. 2020. Performance bug in Util Batch Generator module. <https://github.com/scikit-learn/scikit-learn/issues/16158>. Online.
- [34] Scikit-learn. 2020. Performance bug in Util Batch Generator module fixed. <https://github.com/scikit-learn/scikit-learn/pull/16181/files>. Online.
- [35] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.
- [36] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. *OOPSLA* 49, 10 (2014), 561–578.
- [37] Tensorflow. 2019. Transpose can be very slow on CPU. <https://github.com/tensorflow/tensorflow/issues/27383>. Online.
- [38] Saeid Tizpaz-Niari, Pavol Černý, Bor-Yuh Evan Chang, and Ashutosh Trivedi. 2018. Differential Performance Debugging with Discriminant Regression Trees. In *AAAI*, 2468–2475.
- [39] Saeid Tizpaz-Niari, Pavol Černý, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Discriminating Traces with Time. In *TACAS*. Springer, 21–37.
- [40] Saeid Tizpaz-Niari, Pavol Černý, and Ashutosh Trivedi. 2020. Data-Driven Debugging for Functional Side Channels. <https://arxiv.org/abs/1808.10502>. In *NDSS*.
- [41] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–52.
- [42] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
- [43] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. *The Fuzzing Book*. Saarland University. <https://www.fuzzingbook.org/> Retrieved 2019-09-09 16:42:54+02:00.

# Higher Income, Larger Loan? Monotonicity Testing of Machine Learning Models

Arnab Sharma

Paderborn University

Paderborn, Germany

arnab.sharma@uni-paderborn.de

Heike Wehrheim

Paderborn University

Paderborn, Germany

wehrheim@uni-paderborn.de

## ABSTRACT

Today, machine learning (ML) models are increasingly applied in decision making. This induces an urgent need for *quality assurance* of ML models with respect to (often domain-dependent) requirements. *Monotonicity* is one such requirement. It specifies a software as “learned” by an ML algorithm to give an increasing prediction with the increase of some attribute values. While there exist multiple ML algorithms for *ensuring* monotonicity of the generated model, approaches for *checking* monotonicity, in particular of *black-box* models are largely lacking.

In this work, we propose *verification-based testing* of monotonicity, i.e., the formal computation of test inputs on a white-box model via verification technology, and the automatic inference of this approximating white-box model from the black-box model under test. On the white-box model, the space of test inputs can be systematically explored by a directed computation of test cases. The empirical evaluation on 90 black-box models shows that verification-based testing can outperform adaptive random testing as well as property-based techniques with respect to effectiveness and efficiency.

## CCS CONCEPTS

- Software and its engineering;

## KEYWORDS

Machine Learning Testing, Monotonicity, Decision Tree.

### ACM Reference Format:

Arnab Sharma and Heike Wehrheim. 2020. Higher Income, Larger Loan? Monotonicity Testing of Machine Learning Models. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395363.3397352>

## 1 INTRODUCTION

Today, machine learning (ML) is increasingly employed to take decisions previously made by humans. This includes areas as diverse as insurance, banking, law, medicine or autonomous driving. Hence, quality assurance of ML applications becomes of prime importance.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397352>

Consequently, researchers have started to develop methods checking various sorts of requirements. Depending on the domain of application, such methods target safety, security, robustness or balancedness of ML algorithms and models (e.g., [9, 14, 18, 30]).

A requirement frequently expected in application domains is *monotonicity* with respect to dedicated attributes. Monotonicity requires an increase in the value of some attribute(s) to lead to an increase in the value of the prediction (class attribute). For instance, a loan-granting ML-based software might be required to give larger loans whenever the value of attribute “income” gets higher, potentially even when other attribute values are changed.

Monotonicity requirements occur in numerous domains like economic theory (house pricing, credit scoring, insurance premium determination), medicine (medical diagnosis, patient medication) or jurisdiction (criminal sentencing). In particular, monotonicity is often a requirement for ML software making acceptance/rejection decisions as it supports justification of a decision (“she gets a larger loan because she has got a higher income”). However, even if the training data used to generate the predictive model is monotone, the ML software itself might not be [26]. Hence, there are today a number of specialized ML algorithms which provide learning techniques guaranteeing monotonicity constraints on models (e.g. [26, 34, 36]).

Less studied is, however, the *validation* of monotonicity constraints, i.e., methods for answering the following question:

Given some black-box predictive model, does it satisfy a given monotonicity constraint?

The term “black-box model” states the independence on the ML-technology employed in the model (i.e., the technique should be equally applicable to e.g. neural networks, random forests or support vector machines). We aim at a *model-agnostic* solution.

In this paper, we present the first approach for automatic monotonicity testing of black-box machine learning models. Our approach systematically explores the space of test inputs by (1) the inference of a white-box model *approximating* the black-box model under test (MUT), and (2) the computation of counter examples to monotonicity on the white-box model via established verification technology. We call this approach *verification-based testing*. The computed counter examples serve as starting points for the generation of further test inputs by variation. If confirmed in the black-box model, they get stored as counter examples to monotonicity. If unconfirmed, they serve as input to an improvement of the approximation quality of the white-box model.

More detailedly, our approach comprises the following key steps:

- **White-box model inference.** A white-box model is generated by training a decision tree with data instances of the black-box model under test.

**Table 1: Example banking data set**

No.	income	children	contract	loan
1	100.0	1	20	high
2	25.0	0	2	no
3	17.8	3	5	no
4	25.5	2	15	medium
5	39.0	0	11	medium

- **Monotonicity computation.** The decision tree is translated to a logical formula on which we use an SMT-solver for monotonicity verification.
- **Variation.** The computed counter examples are systematically varied (similar to strategies used in symbolic execution [20]) in order to increase the size of the test suite and its coverage of the test space.
- **White-box model improvement.** In case none of the counter examples are valid for the black-box model under test, we employ the data instances together with the MUT's prediction to re-train the decision tree and thereby restart with an improved white-box model.

We have implemented our approach and have experimentally evaluated it using standard benchmark data sets and both monotonicity aware and ordinary ML algorithms. Our experimental results suggest that our directed generation of test cases outperforms (non-directed) techniques like property-based testing wrt. the effectiveness of finding monotonicity failures.

Summarizing, this paper makes the following contributions:

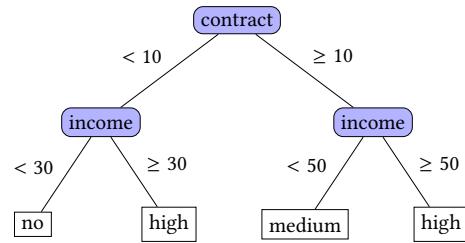
- We formally define monotonicity of ML models.
- We present a novel approach to monotonicity testing via the usage of verification technology on an approximating white-box model.
- We systematically evaluate our approach on 90 black-box models and compare it to state-of-the-art property-based and adaptive random testing.
- For the implementation of adaptive random testing as a baseline (to compare against), we design a distance metric specific to monotonicity testing on ML models.

The paper is structured as follows. In the next section, we define monotonicity of machine learning models. In Section 3 we describe verification-based testing and the way we have implemented adaptive random testing. Section 4 presents the results of our experimental evaluation. We discuss related work in Section 5 and conclude in Section 6.

## 2 MONOTONICITY

We start by introducing the basic terminology in machine learning and defining two notions of monotonicity.

A typical *supervised* machine learning (ML) algorithm works in two steps. Initially, it is presented with a set of data instances called *training data*. In the first (learning) phase, the ML algorithm generates a function (the *predictive model*), generalising from the training data by using some statistical techniques. The generated *predictive model* (short, *model*) is then used in the second (prediction) phase to predict classes for unknown data instances.

**Figure 1: A decision tree for the banking data set**

Formally, the generated model is a function

$$M : X_1 \times \dots \times X_n \rightarrow Y,$$

where  $X_i$  is the value set of *feature i* (or attribute or characteristics  $i$ ),  $1 \leq i \leq n$ , and  $Y$  is the set of *classes*. We define  $\vec{X} = X_1 \times \dots \times X_n$ . The training data consists of elements from  $\vec{X} \times Y$ , i.e., data instances with known associated classes. During the prediction, the generated predictive model assigns a class  $y \in Y$  to a data instance  $(x_1, \dots, x_n) \in \vec{X}$  (which is potentially not in the training data). We assume all  $X_i$  and the set of classes  $Y$  to be equipped with a total order  $\leq_i$  and  $\leq_Y$ , respectively.

In this work, we check whether a given model is *monotone* with respect to a specific feature  $i$ .

**DEFINITION 1.** A model  $M$  is *strongly monotone*<sup>1</sup> with respect to a feature  $i$  if for any two data instances  $x = (x_1, \dots, x_n)$ ,  $x' = (x'_1, \dots, x'_n) \in \vec{X}$  we have  $x_i \leq_i x'_i$  implies  $M(x) \leq_Y M(x')$ .

Note that the feature values apart from the one with respect to which we are checking monotonicity can differ in an arbitrary way. Definition 1 can be weakened as to only require an increasing prediction when all features values apart from the chosen one are kept.

**DEFINITION 2.** A model  $M$  is *weakly monotone* with respect to a feature  $i$  if for any two data instances  $x = (x_1, \dots, x_n)$ ,  $x' = (x'_1, \dots, x'_n) \in \vec{X}$ , we have  $(x_i \leq_i x'_i) \wedge (\forall j, j \neq i. x_j = x'_j)$  implies  $M(x) \leq_Y M(x')$ .

In the literature, the term monotonicity most often refers to our weak version. We also say that a training data set is strongly/weakly monotone if the above requirements hold for all elements in the set. As an example take the training data in Table 1 for a software making decisions about the granting of loans (inspired by [26]): The features of a person are the income (in thousand dollars), the (number of) children and the (duration of) current contract. The class “loan” can take four values: ‘no’, ‘low’, ‘medium’ and ‘high’, with total order ‘no’  $<_Y$  ‘low’  $<_Y$  ‘medium’  $<_Y$  ‘high’. This data set is monotone<sup>2</sup> for features “contract” and “income”, but not for feature “children”.

Figure 1 gives a potential model (in the form of a decision tree) which the training on this data set could yield. It can correctly predict all instances in the training data. However, this model is

<sup>1</sup>Note that “strong” here does not refer to a strong increase in values, i.e., a definition with  $<$  instead of  $\leq$ .

<sup>2</sup>Note that we cannot apply our formal definitions here since the data set does not give us classes for all data instances.

not weakly monotone in feature “contract” anymore: Take e.g. the following two data instances

```
income=30.0, children=0, contract=9
income=30.0, children=0, contract=10
```

While the prediction for the first instance is ‘high’, it is ‘medium’ for the latter.

We next define *group monotonicity* which extends Definitions 1 and 2 to a set of features (called *monotone features*).

**DEFINITION 3.** A predictive model  $M$  is said to be *strongly group monotone* with respect to a set of features  $F = \{i_1, i_2, \dots, i_m\} \subseteq \{1, \dots, n\}$  if for any two data instances  $x = (x_1, \dots, x_n), x' = (x'_1, \dots, x'_n) \in \vec{X}$  we have  $\forall j \in F : x_j \leq_j x'_j$  implies  $M(x) \leq_Y M(x')$ .

Similarly, it is *weakly group monotone* with respect to  $F$  if for all  $x, x'$  we have  $(\forall j \in F : x_j \leq_j x'_j \wedge \forall j \notin F : x_j = x'_j)$  implies  $M(x) \leq_Y M(x')$ .

Strong group monotonicity sits in between weak and strong (single feature) monotonicity in that it allows some feature’s values to change in an arbitrary way while other values may only increase or stay as they are<sup>3</sup>. We see this as being a practically relevant case and have thus included it in our definitions.

Finally, note that test cases for (both strong and weak) monotonicity are by these definitions *pairs* of data instances  $(x, x')$ , and during test execution these need to be checked for the property  $M(x) \leq_Y M(x')$ . If the precondition of the respective monotonicity version holds for  $x$  and  $x'$  but  $M(x) \not\leq_Y M(x')$ , we call the pair  $(x, x')$  a *counter example* to monotonicity.

### 3 TESTING APPROACH

We conduct black-box testing to check monotonicity of the predictive model under test. Hence, in the following we assume the type of the MUT (i.e., which ML algorithm has been used for training) to be unknown.

#### 3.1 Adaptive Random Testing

For the purpose of comparison, we have designed and implemented an *adaptive random testing* (ART) [10] approach for monotonicity testing which we describe first. Adaptive random testing aims at (randomly) computing test cases which are more evenly distributed among the test input space. To this end, it compares new candidates with the already computed test cases, and adds the one “furthest” away. The implementation of “furthest” requires the definition of a *distance metric*. For numerical inputs, this is often the Euclidean distance. For monotonicity, our test cases are however *pairs*  $(x, x')$ .

Assume we are given two such pairs  $(x, x')$  and  $(z, z')$  and want to define how “different” they are. Assume furthermore that all the elements only contain *numerical values*<sup>4</sup>. Every element  $x = (x_1, \dots, x_n)$  can then be considered to be a point in an  $n$ -dimensional space. We let  $Euc(x, x')$  be the Euclidean distance between  $x$  and  $x'$ , and  $m_{x, x'}$  be the point laying at the middle of  $x$  and  $x'$ . The metric which we employ captures two aspects: we see two pairs  $(x, x')$  and  $(z, z')$  as being very different if (a) their Euclidean distances are far apart (e.g.,  $x$  is very close to  $x'$ , but  $z$  far away from  $z'$ ) and (b)

<sup>3</sup>Note that all  $\leq$  orders are reflexive.

<sup>4</sup>This can easily be achieved by some preprocessing step converting categorical to numerical values.

---

**Algorithm 1** artGen (Test Generation for ART)

---

```

Input:  $F$  ▷ set of monotone features
Output: set of test cases
1:  $ts := \emptyset; count := 0;$ 
2: while  $count < \text{INI\_SAMPLES}$  do ▷ randomly generate start set
3:    $x := \text{random}(\vec{X});$ 
4:    $x' := \text{random}(\{x' \mid \forall i \in F : x_i \leq x'_i, \forall j \notin F : x_j = x'_j\});$ 
5:   if  $(x, x') \notin ts$  then
6:      $ts := ts \cup \{(x, x')\}; count++;$ 
7: while  $|ts| < \text{MAX\_SAMPLES}$  do ▷ extend start set
8:    $cand := \emptyset; count := 0;$ 
9:   while  $count < \text{POOL\_SIZE}$  do ▷ generate candidates
10:     $x := \text{random}(\vec{X});$ 
11:     $x' := \text{random}(\{x' \mid \forall i \in F : x_i \leq x'_i, \forall j \notin F : x_j = x'_j\});$ 
12:    if  $(x, x') \notin cand$  then
13:       $cand := cand \cup \{(x, x')\}; count++;$ 
14:     $c_{fur} := \text{oneOf}(cand);$  ▷ initialize with arbitrary cand.
15:     $maxDist := 0;$ 
16:    for  $c \in cand$  do ▷ determine “furthest away” cand.
17:       $dist := \text{minDistance}(c, ts);$ 
18:      if  $dist > maxDist$  then
19:         $c_{fur} := c; maxDist := dist;$ 
20:     $ts := ts \cup \{c_{fur}\};$ 
21: return  $ts;$ 

```

---

the middle of  $(x, x')$  is far away from the middle of  $(z, z')$ . Formally, we define

$$\text{dist}((x, x'), (z, z')) = \frac{|Euc(x, x') - Euc(z, z')|}{2} + \frac{Euc(m_{x, x'}, m_{z, z'})}{2}$$

Note that  $\text{dist}$  is positive-definite, symmetric and subadditive, i.e., indeed a metric. We use this distance function in the test case generation for ART within Algorithm 1 (inspired by a definition of ART algorithms in [35]).

The first loop in Algorithm 1 randomly computes a set of pairs  $(x, x')$  to start with. Note that all these pairs already satisfy the precondition of – in this case – weak monotonicity by construction (line 4). The second (outermost) loop extends this test set until it contains MAX\_SAMPLES pairs. It starts in line 9 by generating a set of candidates. The loop starting in line 16 then determines the candidate “furthest away” from the current test set  $ts$ . It uses the function  $\text{minDistance}$  which computes the minimal distance between  $c$  and elements of  $ts$  using the metric  $\text{dist}$ . The furthest away candidate is put into the test set  $ts$  in line 20 and the algorithm returns with the entire test set in line 21. This test set is then subject to checking all pairs  $(x, x')$  for monotonicity (not given as algorithm here).

For checking strong monotonicity we follow a similar approach with the only exception being the generation of test input pairs  $(x, x')$ . In that case, the changes are in lines 4 and 11 which become  $x' := \text{random}(\{x' \mid x \neq x', (\forall i \in F : x_i \leq x'_i)\})$ .

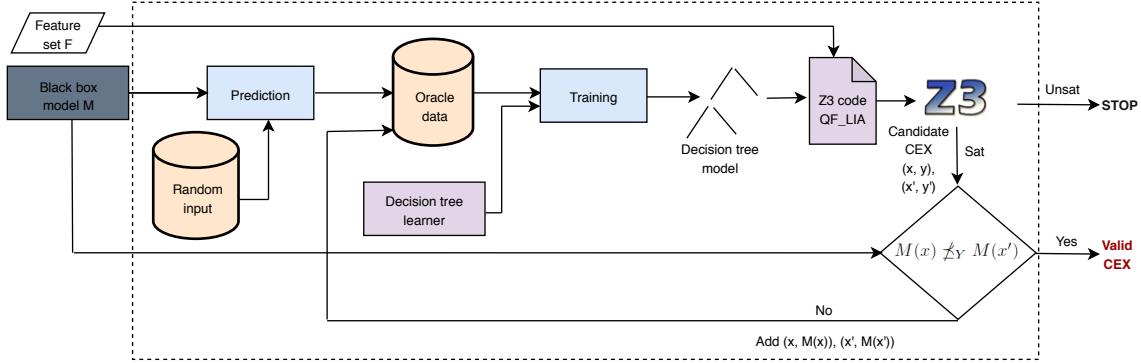


Figure 2: Basic workflow of verification-based testing

### 3.2 Verification-Based Testing

Next, we present our novel approach for the generation of test sets for monotonicity testing of a black-box model. The key idea therein is to approximate the MUT by a white-box model on which verification techniques can *compute* counter examples to monotonicity (in case these exist). These counter examples then serve as test inputs for the MUT, thereby achieving a target-oriented generation of test suites. We call this technique *verification-based testing* (VBT).

The idea of approximating an unknown predictive model by a white-box model is already employed in the areas of interpretability of AI as well as testing of (non-ML) software. In AI (Guidotti et al. [17]) an unknown black-box predictor is converted to an explainable model out of which explanations for humans can be constructed. In the field of testing, Papadopoulos and Walkinshaw [24] use the inference of a predictive model from test sets to further extend this set. None use it to compute counter examples to properties.

In our work, we approximate the black-box MUT by a decision tree. The choice of it is driven by the desire to employ verification-based testing: counter examples to monotonicity are *automatically computable* for decision trees via SMT solvers. Figure 2 depicts the basic workflow; the exact interplay of components is detailed in Algorithm 5. Our approach is composed of four parts.

**White-box model inference.** The inputs to our approach are the predictive model  $M$  (MUT) and a set of features  $F$ . In the first step, we train the decision tree. To this end, we generate so called *oracle data*, containing the predictions of the MUT for some randomly chosen input instances. This set of data instances are currently not used for testing monotonicity (but could be), rather they are employed for the purpose of generating the white-box model. A decision tree learner is then trained on the oracle data giving a decision tree model.

**Monotonicity computation.** Once we have generated the decision tree, the next step is to compute (non-)monotonicity. To this end, we use state of the art verification technology, namely the SMT solver Z3 [12]. First, we translate the decision tree into a logical formula describing how the classes are predicted for inputs  $x$  and  $x'$ . Figure 3 shows the Z3 code for the decision tree in Figure 1. Therein, variables  $contract1$  and  $income1$ <sup>5</sup> describe test input  $x$

```

; Declaring components of x and x' and their classes
(declare-fun contract1 () Int) (declare-fun income1 () Real)
(declare-fun contract2 () Int) (declare-fun income2 () Real)
(declare-fun class1 () Int) (declare-fun class2 () Int)
; Specifying prediction of decision tree (no=0, medium=1, high=2)
(assert (= (and (< contract1 10) (< income1 30)) (= class1 0)))
(assert (= (and (< contract1 10) (>= income1 30)) (= class1 2)))
(assert (= (and (>= contract1 10) (< income1 50)) (= class1 1)))
(assert (= (and (>= contract1 10) (>= income1 50)) (= class1 2)))
(assert (= (and (< contract2 10) (< income2 30)) (= class2 0)))
(assert (= (and (< contract2 10) (>= income2 30)) (= class2 2)))
(assert (= (and (>= contract2 10) (< income2 50)) (= class2 1)))
(assert (= (and (>= contract2 10) (>= income2 50)) (= class2 2)))
; Non-monotonicity constraint
(assert (and (<= contract1 contract2) (= income1 income2)))
(assert (not (<= class1 class2)))
; Satisfiable ?
(check-sat)
; Logical model extraction
(get-model)

```

Figure 3: Z3 code of the decision tree with strong monotonicity constraint

and  $contract2$  and  $income2$  describe  $x'$ . The code also contains the non-monotonicity query for weak monotonicity wrt. "income". The last four lines of the code ask Z3 to check for satisfiability of all assertions and – if yes (Sat) – to return a logical model. The logical model gives an evaluation for the variables such that all assertions are fulfilled. For our example, it can be found in Figure 4. It matches the counter example of Section 2.

The counter example consists of a pair of data instances and their respective classes  $((x, y), (x', y'))$  as predicted by the decision tree. As the approximation of the MUT by the decision tree will typically be imprecise, this counter example might not be valid for the MUT: it is a *candidate* counter example (candidate CEX). Hence, we next check the validity of  $M(x) \not\leq y \wedge M(x') \leq y'$ . If it holds, a true counter example to monotonicity of the MUT (valid CEX) has been found. If not,  $(x, M(x))$  and  $(x', M(x'))$  are added to the oracle data in order to increase precision of the approximation in later steps.

When the output of Z3 is 'Unsat', we conclude that the generated decision tree is monotone wrt.  $F$  (but not necessarily the MUT  $M$ ) and thus verification-based testing has been unable to compute a test case for non-monotonicity.

<sup>5</sup>There is no variable for "children" since the decision tree is not using this feature.

```

sat (model
  (define-fun contract1 () Int 9)
  (define-fun income1 () Real 30.0)
  (define-fun class1 () Int 2)
  (define-fun contract2 () Int 10)
  (define-fun income2 () Real 30.0)
  (define-fun class2 () Int 1))

```

**Figure 4:** Logical model for the query of Fig. 3**Algorithm 2** *prunInst* (Pruning data instances)

**Input:**  $(x, x')$  ▷ A candidate pair  
 $\varphi$  ▷ Logical formula

**Output:** set of candidate pairs

---

```

1: cand-set :=  $\emptyset$ ;
2: for  $i := 1$  to  $n$  do  $n$ : number of features
3:    $\psi := \varphi \wedge \neg(\text{name}_i 1 = x_i)$ ;
4:   if SAT( $\psi$ ) then
5:     cand-set := cand-set  $\cup$  get-model( $\psi$ );
6:   for  $i := 1$  to  $n$  do
7:      $\psi := \varphi \wedge \neg(\text{name}_i 2 = x'_i)$ ;
8:     if SAT( $\psi$ ) then
9:       cand-set := cand-set  $\cup$  get-model( $\psi$ );
10:  return cand-set;

```

---

**Variation.** The basic workflow of Figure 2 is complemented by *variation* techniques. Whenever we obtain a counter example which is not confirmed in the MUT, we need to make the approximating decision tree more precise. This is achieved by re-training. As this is a rather costly operation, we would like to avoid re-training for a single unconfirmed counter example and only re-train once we have collected a number of test pairs. This calls for a systematic generation of counter examples for which we need Z3 to produce several different logical models for the same logical query. To this end, we employ two pruning techniques cutting off certain parts of the search space of Z3 when computing logical models.

**3.2.1 Pruning data instances.** Our first strategy is to call the SMT solver Z3 several times and simply disallow it to return the same counter example again. For our running example, we can simply add (`assert (not (= contract1 9))`) to our query and re-run the SMT solver. We can similarly do so for the values of the other features. This way we can often generate a large number of *similar* counter examples. E.g., for our example Z3 then returns an instance with `contract1` being 8, then 7, and so on. Algorithm 2 describes how we generate new candidate pairs this way. Therein, `namei1` (`namei2`) stands for the name of feature  $i$  in instance  $x$  ( $x'$ , respectively), e.g. `income1`.

The disadvantage of this approach is that Z3 will give counter examples considering only a few number of branches in the decision tree. Hence, a major part of the decision tree paths will remain unexplored. Next we define a second strategy which achieves better coverage of the tree and thereby an improved *test adequacy*.

**3.2.2 Pruning branches.** In this case we use a global approach to traverse as many paths as possible. Once a test pair  $(x, x')$  is found, we identify the paths in the tree which this pair takes. Then we toggle the conditions on  $x$ 's path and on  $x'$ 's path, one after the other.

**Algorithm 3** *prunBranch* (Pruning branches)

**Input:**  $(x, x')$  ▷ A candidate pair  
tree ▷ Decision tree  
 $\varphi$  ▷ Logical formula

**Output:** set of candidate pairs

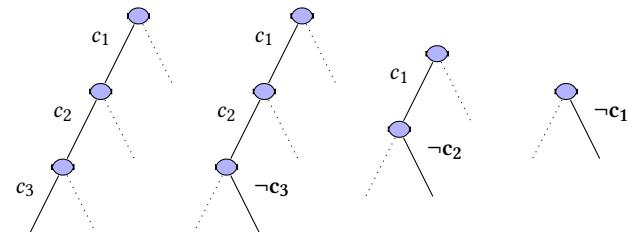
---

```

1: cand-set :=  $\emptyset$ ;
2:  $(c_1, \dots, c_m) := \text{getPath}(\text{tree}, x)$ ; ▷ Path of  $x$  in tree
3: for  $i := 1$  to  $m$  do ▷ toggle path conditions
4:    $\psi := \varphi \wedge \neg c_i$ ;
5:   if SAT( $\psi$ ) then
6:     cand-set := cand-set  $\cup$  get-model( $\psi$ );
7:    $(c_1, \dots, c_k) := \text{getPath}(\text{tree}, x')$ ; ▷ Path of  $x'$  in tree
8:   for  $i := 1$  to  $k$  do ▷ toggle path conditions
9:      $\psi := \varphi \wedge \neg c_i$ ;
10:    if SAT( $\psi$ ) then
11:      cand-set := cand-set  $\cup$  get-model( $\psi$ );
12:  return cand-set;

```

---

**Figure 5:** Illustration of condition toggling

Other toggling strategies are possible. This “condition toggling” follows the established strategy of determining path conditions in symbolic execution and systematically negating conditions for concolic testing [16, 29]. Figure 5 illustrates it on one path (the conditions in bold are added to the Z3 code) and Algorithm 3 gives the algorithm.

While the first pruning strategy tries to find new pairs in the local neighbourhood of a counter example, the second strategy globally searches for counter examples and thus achieves a better coverage. Algorithm 4 summarizes one such pass of counter example candidate generation from the decision tree.

**White-box model improvement.** Once we have collected a larger set of test pairs (all candidate counter examples), we test whether they are also counter examples to monotonicity for the MUT  $M$ . We furthermore check whether  $M$ 's prediction differs from the decision tree's prediction on the candidates. If yes, they will be added to the oracle data to re-train the decision tree.

**Overall algorithm.** Algorithm 5 summarizes all these steps in one algorithm. It interleaves generation of test cases with monotonicity checking, i.e., after one call to *veriGen* it first checks all candidates and only if none of these are counter examples to monotonicity, it starts retraining. Two constants play a rôle in this algorithm: the constant MAX\_ORCL fixes the number of data instances to be generated for initially training the decision tree, and (again) constant MAX\_SAMPLES limits the number of samples we generate<sup>6</sup>.

<sup>6</sup>Note that due to variation,  $ts$  might become slightly larger than MAX\_SAMPLES.

**Algorithm 4** veriGen (Test Generation for VBT)

---

**Input:**  $F$  ▷ Set of features  
     tree ▷ trained decision tree

**Output:** set of test cases

- 1:  $cs := \emptyset;$
- 2:  $\varphi := \text{tree2Logic(tree);}$
- 3:  $\varphi := \varphi \wedge \text{nonMonConstr}(F);$
- 4: **if** UNSAT( $\varphi$ ) **then** ▷ No CEX cand. found
- 5:     **return**  $\emptyset;$
- 6:  $((x, y), (x', y')) := \text{getModel}(\varphi);$  ▷ Gen. candidates
- 7:  $cs := \{(x, y), (x', y')\};$
- 8:  $cs := cs \cup \text{prunInst}(x, x', \varphi) \cup \text{prunBranch}(x, x', \text{tree}, \varphi);$
- 9: **return**  $cs;$

---

**Algorithm 5** veriTest (Verification-based testing)

---

**Input:**  $M$  ▷ Model under test  
      $F$  ▷ Set of features

**Output:** counter example to monotonicity or empty

- 1:  $orcl\_data := \emptyset; ts := \emptyset; cs := \emptyset;$
- 2: **while**  $|orcl\_data| < \text{MAX\_ORCL}$  **do**
- 3:      $x := \text{random}(\bar{X});$
- 4:     **if**  $(x, M(x)) \notin orcl\_data$  **then**
- 5:          $orcl\_data := orcl\_data \cup \{(x, M(x))\};$
- 6: **while**  $|ts| < \text{MAX\_SAMPLES}$  **do**
- 7:      $\text{tree} := \text{trainDecTree}(orcl\_data);$
- 8:      $cs := \text{veriGen}(F, \text{tree});$
- 9:     **for**  $((x, y), (x', y')) \in cs$  **do** ▷ Duplicates?
- 10:         **if**  $(x, x') \notin ts$  **then**
- 11:              $ts := ts \cup \{(x, x')\};$
- 12:         **else**
- 13:              $cs := cs \setminus \{(x, y), (x', y')\};$
- 14:         **if**  $cs = \emptyset$  **then** ▷ No new candidates?
- 15:             **return** Empty;
- 16:         **for**  $((x, y), (x', y')) \in cs$  **do** ▷ Valid CEX?
- 17:             **if**  $M(x) \neq M(x')$  **then**
- 18:                 **return**  $((x, y), (x', y'))$
- 19:             **if**  $y \neq M(x)$  **then** ▷ Different prediction?
- 20:                  $orcl\_data := orcl\_data \cup \{(x, M(x))\};$
- 21:             **if**  $y' \neq M(x')$  **then**
- 22:                  $orcl\_data := orcl\_data \cup \{(x', M(x'))\};$
- 23: **return** Empty; ▷ No counter example found

---

## 4 EVALUATION

We have implemented adaptive random as well as verification-based testing for monotonicity in Python and have comprehensively evaluated VBT. The following research questions guided our evaluation. We broadly divide these questions into two categories. The first category concerns the comparison of VBT with existing techniques with respect to effectiveness and efficiency. The second category evaluates the performance of VBT itself.

### RQ1. Effectiveness

How does VBT compare to existing testing approaches with respect to the error detection capabilities?

**Table 2: Data sets and their characteristics**

Name	#Features	#Group	#Instances	#TreeNodes
Adult	13	4	32561	4673
Diabetes	8	5	768	267
Mammographic	6	3	961	481
Car-evaluation	6	4	1728	167
ESL	4	2	488	295
Housing	13	3	506	107
Automobile	24	10	205	53
Auto-MPG	7	5	392	117
ERA	4	2	1000	87
CPU	6	5	209	11

### RQ2. Efficiency

How does VBT compare to existing testing approaches with respect to the effort for error detection?

To analyse VBT itself we have focused on the following research questions:

### RQ3. Approximation quality

Can the decision trees adequately represent black-box models?

### RQ4. Strategy selection

Which pruning strategy performs better in computing non-monotonicity?

We have carried out the following experiments to evaluate these research questions.

**RQ1.** As there are no specific approaches for computing monotonicity of a given black-box model, we use property-based testing (i.e., a variant of QuickCheck [11] for Python) and (our own implementation of) adaptive random testing as baseline approaches to compare against. Our intention is to measure how well a technique is able to generate test cases revealing non-monotonicity of a given model. To this end, we have taken 8 ML algorithms from the state of the art ML library scikit-learn plus one monotonicity aware classifier [2], and trained them on ten data sets (see below) to generate 90 different predictive models. For these models it is first of all unknown whether they are monotone or not, so we lack a ground truth. The comparison is thus performed on the basis of just counting the number of models in which non-monotonicity is detected<sup>7</sup>. We perform the evaluation for weak (group) monotonicity as it is the standardly employed concept.

**RQ2.** To answer RQ2, we have carried out experiments in the same setting as considered for RQ1. We wanted to evaluate how efficient our verification-based testing approach is compared to adaptive random and property-based testing. To this end, we (a) determine the run time needed for test generation and checking, and (b) the number of generated test cases necessary for finding the first error or, in case of a failure in error detection, just the number of generated tests. In the latter case, this is the maximal number of samples to be considered by the approach, which is configurable for property-based testing and which is MAX\_SAMPLES for ART and VBT (see below for values used).

<sup>7</sup>Note that none of the techniques produce false positives since they all perform a dynamic analysis.

**RQ3.** As we employ decision trees for computing candidate counter examples, the performance of VBT crucially depends on decision trees to adequately approximate the black-box model. “Adequately” here means adequate for the task of computing counter examples. In general, the decision tree model and the black-box will differ on some predictions. The inadequacy of the decision tree shows up whenever we need to re-train it several times in order to find a proper counter example. Hence, for RQ3 we determine the number of re-trainings for all 90 models and weak monotonicity.

**RQ4.** For test generation (Algorithm *veriGen*) we have implemented two different pruning strategies to achieve better coverage of the decision tree. So we wanted to find out which strategy is better in terms of finding counter examples. For the evaluation, we slightly change the setting. First, instead of only using weak monotonicity, we also check for strong monotonicity since we had the impression in initial experiments that the pruning strategies might behave differently for the weak and strong version. Second, we modify VBT such that it generates several counter examples (simply by not stopping it on the first one) and compute the achieved *detection rate*, i.e. number of detected errors divided by number of overall test cases  $\frac{\# \text{errors}}{\# \text{test cases}}$ .

#### 4.1 Setup

We have collected our 10 data sets from the UCI machine learning repository<sup>8</sup> and the OpenML data repository<sup>9</sup>. These training data sets have also been used in existing works [21, 34] on monotonicity. Table 2 shows the data sets and their characteristics, i.e., the number of features, the size of the group (number of features in the group) and the number of data instances in the set. We have also computed the number of nodes of the decision tree model (column #TreeNodes) when being trained on the corresponding dataset to give a rough idea about the size of decision trees generated in VBT.

The *monotone features* are chosen based on our own assumptions about the domain and previous works. For instance, in case of the Adult data set, where a model predicts whether a person’s income is at least \$50,000, we check monotonicity with respect to the group consisting of age, weekly working hours, capital-gain and education level [36].

The eight classification algorithms which have been taken from scikit-learn are kNN, Neural Networks (NN), Random Forests (RF), Support Vector Machines (SVM), Naive Bayes (NB), AdaBoost, GradientBoost and Logistic Regression. We have used a linear kernel for SVM and a Gaussian version of NB for our experiments. There are several other ML algorithms which can be found in the library but the eight algorithms chosen here belong to the most basic family of ML classifiers. The 9th ML classifier is the monotonicity aware algorithm LightGBM [2] which has been specifically designed to construct models being monotone with respect to a given set of features. The monotonicity constraints can be enforced during the training phase of this algorithm. This should guarantee monotonicity but – as initial experiments have revealed – LightGBM does not entirely manage to rule out non-monotonicity. This classifier is an excellent benchmark for the three approaches since

<sup>8</sup><https://archive.ics.uci.edu/ml>

<sup>9</sup><https://www.openml.org>

**Table 3: Number of non-monotonicity detections**

Classifiers	VBT	ART	PT
k-NN	9	9	7
Logistic Regression	8	8	6
Naive Bayes	7	4	5
SVM	9	8	5
Neural Network	8	6	4
Random Forest	9	9	5
AdaBoost	8	7	5
GradientBoost	8	7	5
LightGbm	2	0	0
Overall	68	58	42

there are only a very few erroneous input pairs and the challenge is to generate exactly these as test cases.

We have evaluated the accuracy score while generating predictive models and used the score to adjust the hyperparameters of the learning algorithms. While generating our white-box model (i.e. decision tree), we use the default hyperparameter settings from scikit-learn. This essentially does not fix the depth of the tree and hence causes the tree to overfit to the training data (which in our case is the oracle data). As we try to bring the decision tree as close to the black box model as possible, this is actually an advantage, not a disadvantage.

The input parameters of *artGen* and *veriTest* algorithms have been chosen based on execution time of some initial experiments. The parameter POOL\_SIZE of the *artGen* algorithm is set to half of INI\_SAMPLES which is 100; for MAX\_SAMPLES we use 1000.

We have created oracle data in the verification-based testing approach by generating random data instances (90%) and also taking training data instances randomly (10%)<sup>10</sup>. This choice is influenced by the work of Johansson et al. [19] who found that using random data instances to approximate a model gives the best result.

We have used HYPOTHESIS [1], a Python version of QuickCheck [11] as our property-based testing tool. Property-based testing allows users to specify the property to be tested. The parameters of this tool have been set in accordance with the *artGen* and *veriTest* algorithms (parameter for upper bound of test cases is 1000 like MAX\_SAMPLES).

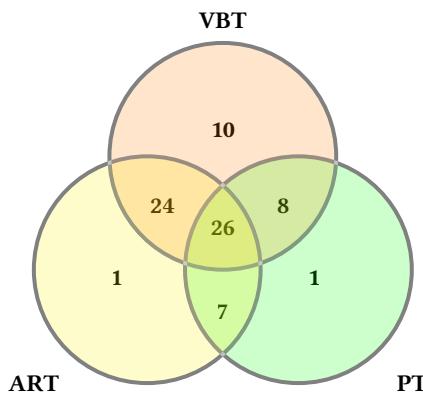
Finally, because all three approaches involve some sort of randomness, every experiment was carried out ten times. The results give the arithmetic mean over these ten runs. The experiments were run on a machine with 2 cores Intel(R) Core(TM) i5-7300U CPU with 2.60GHz and 16GB memory using Python version 3.6.

#### 4.2 Results

Next, we report on the findings of our experiments while evaluating the research questions. All the necessary code and the datasets required to replicate the results reported here are available at <https://github.com/arnabsharma91/MonotonicityChecker>.

**RQ1 - Effectiveness.** Table 3 shows the results of the experiments for RQ1. It gives the number of models (out of 90 in total) for which our approach verification-based testing (VBT) and the two baselines adaptive random testing (ART) and property-based

<sup>10</sup>Note that for simplicity the stated algorithm does not include the training part.



**Figure 6: Venn diagram showing distribution of detected non-monotone models on approaches**

testing (PT) were able to detect non-monotonicity. Note again that the ground truth, i.e., which models are in fact non-monotone, is unknown, but all reported non-monotonicity cases are true positives. Per classifier 10 models were tested.

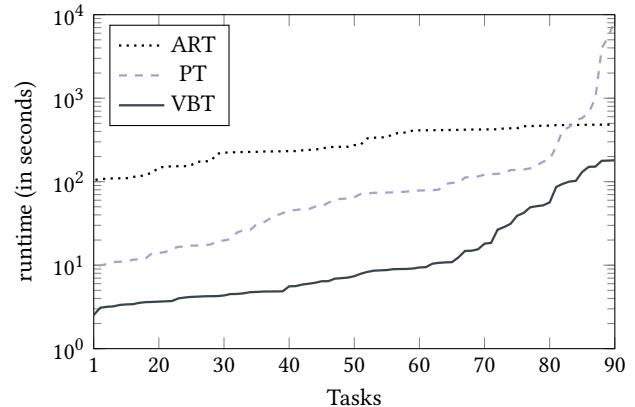
The results show that verification-based testing is more effective in detecting non-monotonicity than both adaptive random and property-based testing. It also shows that adaptive random testing can outperform (pure) property-based testing because – with the help of the distance metric – it more systematically generates test cases covering the test input space.

Another interesting result is the detection of non-monotonicity in two (out of 10) models generated by LightGBM (and the fact that ART and PT fail to detect it). LightGBM is a monotonicity-aware classifier which is supposed to just generate monotone models. For two of the training sets it has however failed to do so, resulting in a model which still has a small number of non-monotone pairs which VBT can find, but ART and PT cannot.

Given these differences in numbers, we also wanted to know whether the non-monotonicity detections of ART and PT are simply a subset of those of VBT. This is actually not the case. The Venn diagram in Figure 6 shows the distribution of counter examples onto the three approaches. 26 models are in the intersection of all three techniques. For the rest, only one or two of the approaches could detect non-monotonicity. The diagram also shows that there are 9 models for which both PT or ART can detect non-monotonicity, but VBT cannot. These models are all models trained on the ERA (6 models) and ESL (3 models) data sets. Looking at the models themselves, it turns out that their accuracy (wrt. the training data) is always very low (below 0.62 and for ERA even below 0.3). Hence, it seems that generalization from this training data is difficult for ML algorithms, and the decision trees in VBT seem to generalize in a different way than the black-box models and hence approximate them less well. This also shows in the results of RQ3 below concerning the ERA and ESL data sets.

Summarizing the findings of RQ1 in our experiments, we get

On average, VBT is more effective than ART and PT in detecting non-monotonicity of black-box ML models.



**Figure 7: Run time in checking monotonicity**

**RQ2 - Efficiency.** For RQ2, we designed experiments to evaluate how efficient VBT is in comparison to ART and PT. Figure 7 shows the runtime of the three approaches for testing monotonicity. The x-axis enumerates the 90 tasks (i.e., models to be tested) where the tasks are sorted in ascending order of runtime per approach, and the y-axis gives the runtime for testing the task (in seconds, on a logarithmic scale). It shows that for all tasks VBT takes less time than both ART and PT even though our approach consists of several steps (including the training of a decision tree).

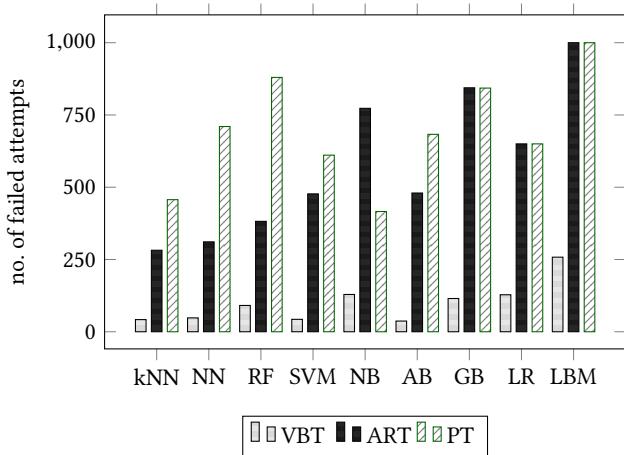
Adaptive random testing takes (on average) the same amount of time for all tasks. During ART, most of the time is needed for creating the input test cases which are “furthest” away from each other. As the size of test inputs is always the same for all the test cases, the time does not vary that much. On the other hand, property-based testing performs better than ART in most of the cases apart from some exceptions.

Second, for RQ2 we have determined the number of test cases generated during testing. All three approaches stop once they have detected the first counter example to monotonicity. Figure 8 shows the number of failed attempts (i.e., generated test cases before finding the first counter example) for each of the classifiers averaging over all the datasets. Our experimental results suggest that VBT always needs the least amount of test cases to testify non-monotonicity.

Note that our approach has two possible execution instances when not finding counter examples: (1) the SMT solver might continuously generate counter example candidates which all fail to be real counter examples in the model and thus VBT successively retrains the tree until MAX\_SAMPLES candidates have been generated, or (2) it immediately stops because the first decision tree is already monotone and the SMT solver finds no counter example at all. In the latter case, the number of failed attempts is 0 (which is favourable for a low number of attempts). However, this only occurred in 7 out of the 90 models.

Summarizing the findings of RQ2 in our experiments, we get

On average, VBT is more efficient than ART and PT in detecting non-monotonicity of black-box ML models.

**Figure 8: Number of failed attempts**

For the next two research questions we look at verification-based testing only.

**RQ3 - Approximation quality.** Table 4 shows the mean number of re-trainings of the decision tree per classifier and data set (mean over 10 runs). For those cases where VBT could not find any non-monotonicity, we have written '-'. As the results show, the number of re-trainings is high for the monotonicity aware algorithm LightGBM (LBM), which is consistent with the results shown in Figure 8. Apart from the classifier, the data set seems to influence the number of retrainings (e.g., Adult needs a large number of retrainings). In general, the numbers are – however – relatively low ( $\leq 10$ ). Note that there are also 27 models for which *no* retraining at all is needed.

Hence, we conclude the following.

On average, the approximation quality of decision trees in VBT is good enough to only require a small number of retrainings for non-monotonicity detection.

**RQ4 - Strategy Selection.** For RQ4, we modified VBT as to not stop upon the first valid counter example. Figures 9 and 10 show the *detection rates* (number of detected counter examples divided by number of test cases) of the two pruning strategies alone in computing strong (Fig. 9) and weak monotonicity (Fig. 10), respectively. Note the difference in the maximal detection rate which is 0.5 for strong and only 0.25 for weak monotonicity. Our experimental results suggest that on a large number of classifiers branch pruning is better or equal to feature pruning for strong monotonicity. On the other hand, for weak monotonicity this is the opposite. This can partly be explained on the decision tree itself: since weak monotonicity requires all but the values of monotone features to be the same in a pair, branch pruning cannot exhibit its full power.

The results suggest the following.

**Table 4: Mean number of re-trainings**

Classifier \ Data	kNN	NN	RF	SVM	NB	AB	GB	LR	LBM
Adult	4	6	19	7	26	1	25	37	-
Auto	0	1	3	0	-	3	2	0	35
Car	0	0	0	0	0	-	1	5	-
CPU	0	0	1	0	26	0	9	1	-
Diabetes	3	0	1	0	4	8	2	7	-
ERA	1	-	-	-	-	0	-	-	-
ESL	-	-	1	0	-	-	-	-	-
Housing	0	0	1	0	1	0	1	0	-
Mammo	1	0	0	1	1	0	1	5	-
Mpg	0	10	0	1	0	0	0	5	18

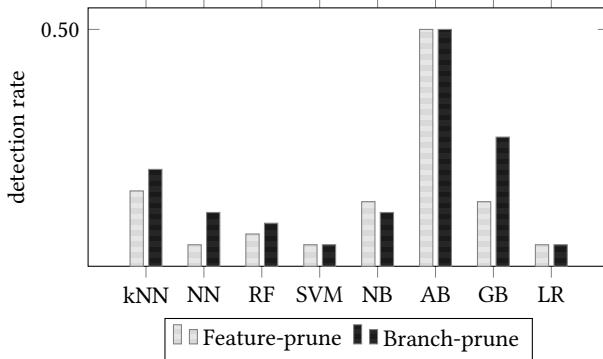
On average, branch pruning achieves a higher detection rate than feature pruning for strong monotonicity and vice versa for weak monotonicity.

### 4.3 Limitations and Threats to Validity

Since we employ an SMT solver for monotonicity computation, verification-based testing is restricted to feature values allowed by the solver. Currently, we have data sets with integer and real values. For other domains of feature values, an encoding would be necessary. This is however often done by ML algorithms within preprocessing steps anyway, so we could easily make use of existing techniques there.

Threats to the validity of results are the choice of data sets and ML algorithms and the choice of feature groups for monotonicity checking. For the ML algorithms, we are confident that we have covered all sorts of basic classifiers in usage today (of course, there are in addition numerous specialised classifiers which however often make use of the base techniques). As we have taken a number of different, publicly available data sets for machine learning, we are furthermore confident that our data sets are diverse enough to exhibit different properties of the approaches and reflect real data sets. In particular, the ERA data set exposes interesting properties of verification-based testing.

A threat to the internal validity is the high degree of randomness involved in the techniques. First, a number of classifiers use randomized algorithms for generating models. Thus, in principle we might get one monotone and one non-monotone model when training *with the same classifier on exactly the same data set*. To ensure fairness during comparison, all three approaches were always started with the same model as input (training of MUTs is external to testing). Second, all three approaches themselves randomly generate (at least some) data instances (ART and PT as potential test cases, VBT for oracle data). VBT in addition uses a decision tree training algorithm which itself involves randomness. Hence, our decision trees can vary from one run to the next, and this stays so even if we would fix the oracle data. To mitigate these threats, all experiments were performed 10 times and the results give the mean over these 10 runs.



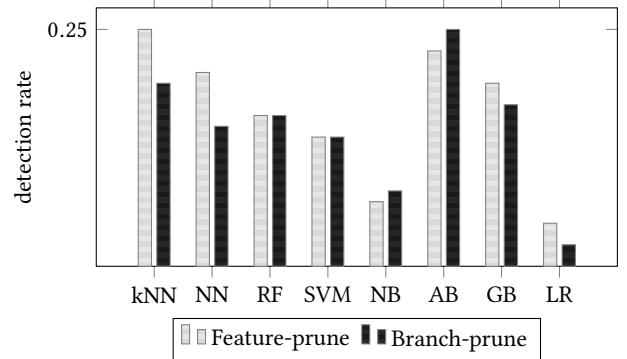
**Figure 9: Performance of branch and feature pruning in computing strong monotonicity**

## 5 RELATED WORK

We divide our discussion of related works in three parts. First, we discuss some works incorporating monotonicity in the predictive model, then mention some recent techniques for machine learning testing and third discuss approaches using model inference in testing.

*Generating monotone models.* The existing works in monotonicity focus on specific ML algorithms. In [6], Archer et al. first propose building a monotone neural network model by adjusting the contribution of training samples in the training process. There exists some follow up works which constrain the parameters of the neural network algorithm to enforce monotonicity [13, 32]. In a more recent work You et al. [36] propose an approach to generate a guaranteedly monotone deep lattice network with respect to a given set of features. Lauer et al. [22] enforce monotonicity in support vector machines (with linear kernels) by constraining the derivative to be positive within a specified range. Riihimäki et al. [28] build a Gaussian monotone model by using virtual derivative observations. In a follow up work, Siivila et al. [31] give an approach based on the same idea to detect monotonicity only for Gaussian distributions. Although using derivatives of the function can work for some ML algorithms, it cannot be generalized and is not possible to use in algorithms where the learned functions (i.e., models) are non-linear.

*Validating models.* There are a number of recent works which aim at validating properties of predictive models, none of which however have looked at monotonicity. In [18], Huang et al. propose robustness as a safety property and give a verification technique showing that a Deep Neural Network (DNN) guarantees postconditions to hold on its outputs when the inputs satisfy a given precondition. Gehr et al. [15] use abstract interpretation to verify robustness of DNN models. Pei et al. [25] propose the first white-box testing technique to test DNNs. They use neuron coverage as a criterion to generate the test cases for the predictive model. Sun et al. [33] propose concolic testing to test the robustness property of DNNs. The authors use a set of coverage requirements (such as neuron, MC/DC and neuron boundary coverage, Lipschitz continuity) to generate test inputs.



**Figure 10: Performance of branch and feature pruning in computing weak monotonicity**

Recently, Sharma et al. [30] have proposed a property called *balancedness* on the learning algorithm. They perform specific transformations on the training data and check whether the learning algorithm generates a different predictive model after applying such transformations. This work thus focusses on testing the ML algorithm itself, not the model.

In [14], Galhotra et al. perform black-box testing to check *fairness* of the predictive model. Basically, they use random testing with confidence driven sampling. It has the drawback of generating completely random sets of test data without considering the structure of the model.

The work closest to us is that of Agarwal et al. [3]. They also study fairness testing, but compared to Galhotra et al. [14] they aim at a more systematic generation of test inputs. To this end, they employ LIME [27] (a tool for generating *local* explanations for predictions) to generate a *partial* decision tree (often just a path in the tree) from the black-box model. On this path, they use dynamic symbolic execution to generate multiple test cases, much alike we do. The difference to our work is that we generate a decision tree approximating the *entire* black-box model under test, and – more importantly – we use the generated white-box model for the *computation* of test inputs (potential counter examples to monotonicity). We thus achieve a targeted test case generation.

*Testing via model inference.* The use of learning in testing has long been considered in the field of model-based testing. Therein, learning is used to extract a model of the system under test. Such models most often are some sort of automaton (finite state machine) and learning is based on Angluin's L\* algorithm [5]. For a survey of techniques see [4, 23].

In contrast to this, we employ machine learning techniques to infer a model. The inference of a decision tree describing the behaviour of software has already been pursued by Papadopoulos and Walkinshaw [24] as well as Briand et al. [8]. The former – similar to us – translate the decision tree to logic in order to have Z3 generate test inputs covering different branches of the tree. However, they do not employ the tree to generate counter examples to the property to be tested. Thus, the advantage of having a verifiable white-box model for targeted test input generation is not utilized. Briand et

al. on the other hand use the decision tree in a semi-automated approach to the re-engineering of test suites. This approach requires the manual inspection of the decision tree by testers. A survey on inference-driven techniques is given in [35].

## 6 CONCLUSION

In this work, we have defined the property of monotonicity of ML models and have proposed a novel approach to testing monotonicity. Our technique approximates the black-box model by a white-box model and applies SMT solving techniques to compute monotonicity on the white-box model. We have evaluated the effectiveness and efficiency of our approach by applying it to several ML models and found our approach to outperform both adaptive random and property-based testing.

As future work, we plan to apply this scheme to validate other important properties of ML models. Our white-box model easily allows for checking other properties, like for instance fairness, just by applying a different check on the generated SMT code. Also, we would like to improve our framework by using incremental learning to avoid re-training of the entire decision tree.

## ACKNOWLEDGEMENTS

We would like to thank Vitalik Melnikov and Eyke Hüllermeier for several discussions on monotonicity in machine learning, and Cedric Richter for his feedback on the experimental evaluation.

## REFERENCES

- [1] 2019. Hypothesis. <https://github.com/HypothesisWorks/hypothesis>. (2019).
- [2] 2019. LightGBM. <https://github.com/Microsoft/LightGBM>. (2019).
- [3] Aniya Aggarwal, Pranay Lohia, Seema Nagar, Kuntal Dey, and Diptikalyan Saha. 2019. Black box fairness testing of machine learning models. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. 625–635. <https://doi.org/10.1145/3338906.3338937>
- [4] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. 2018. Model Learning and Model-Based Testing, See [7], 74–100. [https://doi.org/10.1007/978-3-319-96562-8\\_3](https://doi.org/10.1007/978-3-319-96562-8_3)
- [5] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
- [6] Norman P Archer and Shouhong Wang. 1993. Application of the back propagation neural network algorithm with monotonicity constraints for two-group classification problems. *Decision Sciences* 24, 1 (1993), 60–75.
- [7] Amel Bennaceur, Reiner Hähnle, and Karl Meinke (Eds.). 2018. *Machine Learning for Dynamic Software Analysis*. Lecture Notes in Computer Science, Vol. 11026. Springer. <https://doi.org/10.1007/978-3-319-96562-8>
- [8] Lionel C. Briand, Yvan Labiche, Zaheer Bawar, and Nadia Traldi Spido. 2009. Using machine learning to refine Category-Partition test specifications and test suites. *Information & Software Technology* 51, 11 (2009), 1551–1564. <https://doi.org/10.1016/j.infsof.2009.06.006>
- [9] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy, SP*. 39–57. <https://doi.org/10.1109/SP.2017.49>
- [10] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *ASIAN (Lecture Notes in Computer Science)*, Michael J. Maher (Ed.), Vol. 3321. Springer, 320–329. [https://doi.org/10.1007/978-3-540-30502-6\\_23](https://doi.org/10.1007/978-3-540-30502-6_23)
- [11] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *(ICFP '00)*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. <https://doi.org/10.1145/351240.351266>
- [12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. 337–340. [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
- [13] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. 2009. Incorporating Functional Knowledge in Neural Networks. *J. Mach. Learn. Res.* 10 (2009), 1239–1262. <https://dl.acm.org/citation.cfm?id=1577111>
- [14] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 498–510.
- [15] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *IEEE Symposium on Security and Privacy, SP*. 3–18. <https://doi.org/10.1109/SP.2018.00058>
- [16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. 213–223. <https://doi.org/10.1145/1065010.1065036>
- [17] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2019. A Survey of Methods for Explaining Black Box Models. *ACM Comput. Surv.* 51, 5 (2019), 93:1–93:42. <https://doi.org/10.1145/3236009>
- [18] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV*. 3–29. [https://doi.org/10.1007/978-3-319-63387-9\\_1](https://doi.org/10.1007/978-3-319-63387-9_1)
- [19] Ulf Johansson and Lars Niklasson. [n. d.]. Evolving decision trees using oracle guides. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2009*. 238–244. <https://doi.org/10.1109/CIDM.2009.4938655>
- [20] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. <https://doi.org/10.1145/360248.360252>
- [21] Wojciech Kotłowski and Roman Słowiński. 2009. Rule learning with monotonicity constraints. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009*. 537–544. <https://doi.org/10.1145/1553374.1553444>
- [22] Fabien Lauer and Gérard Bloch. 2008. Incorporating prior knowledge in support vector regression. *Machine Learning* 70, 1 (2008), 89–118. <https://doi.org/10.1007/s10994-007-5035-5>
- [23] Karl Meinke. 2018. Learning-Based Testing: Recent Progress and Future Prospects, See [7], 53–73. [https://doi.org/10.1007/978-3-319-96562-8\\_2](https://doi.org/10.1007/978-3-319-96562-8_2)
- [24] Petros Papadopoulos and Neil Walkinshaw. 2015. Black-Box Test Generation from Inferred Models. In *RAISE*, Rachel Harrison, Ayse Basar Bener, and Burak Turhan (Eds.). IEEE Computer Society, 19–24. <https://doi.org/10.1109/RAISE.2015.11>
- [25] Kexin Pei, Yinzhí Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 1–18. <https://doi.org/10.1145/3132747.3132785>
- [26] R. Pothast and A. J. Feelders. 2002. Classification Trees for Problems with Monotonicity Constraints. *SIGKDD Explor. Newsl.* 4, 1 (June 2002), 1–10. <https://doi.org/10.1145/568574.568577>
- [27] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. “Why Should I Trust You?”, Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1135–1144. <https://doi.org/10.1145/2939672.2939778>
- [28] Jaakkko Riihimäki and Aki Vehtari. 2010. Gaussian processes with monotonicity information. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS*. 645–652. <http://proceedings.mlr.press/v9/riihimaki10a.html>
- [29] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 263–272. <https://doi.org/10.1145/1081706.1081750>
- [30] Arnab Sharma and Heike Wehrheim. 2019. Testing Machine Learning Algorithms for Balanced Data Usage. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST*. 125–135. <https://doi.org/10.1109/ICST.2019.00022>
- [31] Eero Siivola, Juho Piironen, and Aki Vehtari. 2016. Automatic monotonicity detection for Gaussian Processes. *arXiv preprint arXiv:1610.05440* (2016).
- [32] Joseph Sill. 1997. Monotonic Networks. In *Advances in Neural Information Processing Systems*. 661–667. <http://papers.nips.cc/paper/1358-monotonic-networks>
- [33] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE*. 109–119. <https://doi.org/10.1145/3238147.3238172>
- [34] Ali Fallah Tehrani, Weiwei Cheng, Krzysztof Dembczynski, and Eyke Hüllermeier. 2011. Learning Monotone Nonlinear Models Using the Choquet Integral. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML*. 414–429. [https://doi.org/10.1007/978-3-642-23808-6\\_27](https://doi.org/10.1007/978-3-642-23808-6_27)
- [35] Neil Walkinshaw. 2018. Testing Functional Black-Box Programs Without a Specification, See [7], 101–120. [https://doi.org/10.1007/978-3-319-96562-8\\_4](https://doi.org/10.1007/978-3-319-96562-8_4)
- [36] Seungil You, David Ding, Kevin Robert Canini, Jan Pfeifer, and Maya R. Gupta. 2017. Deep Lattice Networks and Partial Monotonic Functions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*. 2985–2993. <http://papers.nips.cc/paper/6891-deep-lattice-networks-and-partial-monotonic-functions>

# Detecting Flaky Tests in Probabilistic and Machine Learning Applications

Saikat Dutta

University of Illinois  
Urbana, IL, USA  
saikatd2@illinois.edu

Zhekun Zhang

University of Illinois  
Urbana, IL, USA  
zhekunz2@illinois.edu

August Shi

University of Illinois  
Urbana, IL, USA  
awshi2@illinois.edu

Aryaman Jain

University of Illinois  
Urbana, IL, USA  
aryaman4@illinois.edu

Rutvik Choudhary

University of Illinois  
Urbana, IL, USA  
rutvikc2@illinois.edu

Sasa Misailovic

University of Illinois  
Urbana, IL, USA  
misailo@illinois.edu

## ABSTRACT

Probabilistic programming systems and machine learning frameworks like Pyro, PyMC3, TensorFlow, and PyTorch provide scalable and efficient primitives for inference and training. However, such operations are non-deterministic. Hence, it is challenging for developers to write tests for applications that depend on such frameworks, often resulting in *flaky tests* – tests which fail non-deterministically when run on the same version of code.

In this paper, we conduct the first extensive study of flaky tests in this domain. In particular, we study the projects that depend on four frameworks: Pyro, PyMC3, TensorFlow-Probability, and PyTorch. We identify 75 bug reports/commits that deal with flaky tests, and we categorize the common causes and fixes for them. This study provides developers with useful insights on dealing with flaky tests in this domain.

Motivated by our study, we develop a technique, *FLASH*, to systematically detect flaky tests due to assertions passing and failing in different runs on the same code. These assertions fail due to differences in the sequence of random numbers in different runs of the same test. *FLASH* exposes such failures, and our evaluation on 20 projects results in 11 previously-unknown flaky tests that we reported to developers.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Probabilistic Programming, Machine Learning, Flaky tests, Randomness, Non-Determinism

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397366>

## ACM Reference Format:

Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397366>

## 1 INTRODUCTION

With the surge of machine learning, randomness is becoming a common part of a developer's workflow. For instance, various training algorithms in machine learning use randomness (in data collection, observation order, or the algorithm itself) to improve their generalizability [21, 33, 60]. As another example, probabilistic programming [9, 10, 16, 25, 66] is an emerging framework for expressive Bayesian modeling with efficient inference. Most existing inference algorithms, such as Markov Chain Monte Carlo [44] and Stochastic Variational Inference [7] are inherently randomized.

The traits of algorithms in various machine learning applications, including inherent randomness, probabilistic specifications, and the lack of solid test oracles [5], pose significant challenges for testing these applications. Recent studies identify multiple classes of domain specific errors in frameworks for both deep learning [53] and probabilistic programming [18]. For instance, probabilistic programs provide distributions as outputs instead of individual values. Developers of projects that use probabilistic programming systems typically have to design tests that run an inference algorithm and check whether the outputs fall within some reasonable range or differ by only a small amount from the expected result. However, determining these thresholds or how many iterations to run is often non-intuitive and subject to heuristics, which may be either too liberal (e.g., assuming independence when one may not exist) or too conservative (e.g., running programs too many times).

As a result of non-systematic testing in this domain, many tests in machine learning and probabilistic programming applications end up being *flaky*, meaning they can non-deterministically pass or fail when run on the same version of code [8, 40, 65]. Furthermore, as developers evolve their code, they rely on regression testing, which is the practice of running tests after every change to check that said changes do not break existing functionality [47, 70]. Regression testing becomes more challenging if the tests can pass and fail even without any changes to the code.

Luo et al. previously investigated flaky tests in traditional software [40]. They identified various reasons for flaky tests and found that some of the most common causes include async wait, concurrency, test-order dependencies, and randomness. However, these studies did not investigate the flaky tests in projects that use probabilistic programming systems or machine learning frameworks, where the inherent probabilistic nature of such systems can yield a different spectrum of reasons for flaky tests in dependent projects, or even new reasons altogether. Furthermore, the way developers fix or mitigate flaky tests in these domains will also differ from how developers of traditional software address them.

**Our Work.** We present a technique for detecting flaky tests in projects using probabilistic programming systems and machine learning frameworks. While common wisdom would suggest that many problems with programs dealing with randomness could be solved by simply fixing the seed of the random number generator, this paper shows that fixing the seed is not always the best solution. Moreover, fixing the seed may not be sufficient for identifying bugs and can be brittle in the presence of program changes.

We conduct *the first study of the common causes and fixes for flaky tests in projects that build upon and use probabilistic programming systems and machine learning frameworks*. We perform an extensive study on 345 projects that depend on four of the most common open-source probabilistic programming systems and machine learning frameworks. We identify 75 bug reports and commits across 20 projects where the developers explicitly fix some flaky tests. We categorize these fixes based on (1) the cause of flakiness and (2) the fix patterns. Unlike Luo et al. [40], we find that projects that depend on probabilistic programming systems and machine learning frameworks have a *larger percentage* of flaky tests whose cause is what Luo et al. would refer to as *randomness*.

We do a more thorough analysis of the flaky tests due to randomness in this domain, breaking them down into more specific categories relevant to probabilistic programming systems and machine learning frameworks. We find that the majority of the causes for flaky tests (45 / 75) are due to Algorithmic Non-determinism, where tests have different results due to the underlying inference mechanism and sampling using probabilistic programming systems and machine learning frameworks. We also find that the most common fix pattern for flaky tests is to adjust the thresholds for the assertions that have flaky failures.

As developers are fixing flaky tests by adjusting their assertions' threshold, we develop **FLASH**, *a novel technique for systematically detecting tests that are flaky due to incorrectly set thresholds in the assertions*. These assertions fail due to differences in the sequence of random numbers in different runs of the same test. FLASH exposes such failures by running the tests using different seeds. FLASH reports which seeds lead to runs where an assertion fails, allowing the user to reproduce such failures.

A distinctive feature of FLASH is its use of statistical convergence tests to identify potentially flaky test executions. FLASH dynamically determines how many times to run each test and assertion with different seeds by sampling the actual values computed for each assertion and running a statistical test to determine if the sampled values have converged, indicating that FLASH has likely seen enough samples to form a distribution of the values. FLASH

then reports the distribution of sampled actual values, providing the user with information on how to fix the flakiness in the test.

We run FLASH on the 20 projects that historically had flaky tests, as we found from our study. FLASH detects 11 previously unknown flaky tests on the latest version of these projects, and we submit 4 patches to fix 5 flaky tests and 6 bug reports for the remaining flaky tests to the developers. Developers confirmed and accepted our fixes for 5 flaky tests and confirmed 5 other flaky tests that are still pending fixes. The remaining 1 bug report is awaiting developer response. We also run FLASH on the historical versions of each project to target previously known flaky tests. We are able to detect 11 such flaky tests using FLASH.

**Contributions.** The paper makes several main contributions:

- We perform the first empirical study on flaky test causes and fixes for projects that depend on probabilistic programming systems and machine learning frameworks. We investigate 75 fixes for flaky tests within 20 projects, where the most common cause for flakiness is due to Algorithmic Non-determinism and the most common fix is to adjust the flaky assertion's threshold.
- We propose FLASH, a technique for systematically detecting flaky tests that fail due to differences in the sequences of random numbers required by computations running through a probabilistic programming system or machine learning framework.
- Our evaluation of using FLASH on 20 projects results in 11 detected flaky tests, and we submit 5 fixes and 6 bug reports for these flaky tests to developers. Developers confirmed and accepted 5 fixes and confirmed 5 other flaky tests. We also detect 11 previously known historical flaky tests using FLASH.

Our datasets and tool for this paper are available for open-access at <https://github.com/uiuc-arc/flash>.

## 2 EMPIRICAL STUDY

The goal of our empirical study is to understand the common causes and fixes for flaky tests in projects that depend on probabilistic programming systems and machine learning frameworks.

### 2.1 Evaluation Projects

**Table 1: Project Statistics**

	Pyro	PyMC3	TF-Prob.	PyTorch
First Commit	Jun 15 '17	Apr 13 '12	Feb 13 '18	Jan 25 '12
#Contributors	73	227	106	1246
#Commits	1823	7223	2254	23,263
#Dependents	4	24	27	290
Prog. Lang.	Python	Python	Python	Python, C++

Initially, we determine four well-known and commonly used probabilistic programming systems and machine learning frameworks: Pyro [9], PyMC3 [59], TensorFlow Probability [17], and PyTorch [51]. We choose these systems because they are open source, have a relatively long development history, and have a large user base. Table 1 presents the statistics for these four frameworks. The table shows for each framework its first commit date, number of contributors, number of commits up until Dec 20, 2019, number

**Table 2: Details of Flaky Test Fixes in Dependent Projects**

Project	Dependent Projects	Filtered Bug Reports	Filtered Commits
Pyro	1	8	7
PyMC3	1	0	2
TensorFlow-Prob	9	12	29
PyTorch	65	191	184
<b>Total</b>	<b>76</b>	<b>211</b>	<b>222</b>
<b>With “test” keyword</b>	<b>33</b>	<b>110</b>	<b>105</b>
<b>Manual Inspection</b>	<b>20</b>	<b>38</b>	<b>37</b>

of dependent projects<sup>1</sup> with more than 10 stars on GitHub, and the major programming language. For PyTorch, we list both Python and C++, because the core library is built using C++, but the API and several utilities are designed using Python.

## 2.2 Extracting Bug Reports and Commits

We start with the 345 projects (summing up the row #Dependents in Table 1) that depend on a probabilistic programming system or machine learning framework. We collect each project’s bug reports (both the Issue Requests and Pull Requests on GitHub) and commits. We filter these bug reports/commits by searching for the following keywords on the bug reports’ conversation text and commit messages: flaky|flakey|flakiness|intermit|fragile|brittle. Next, we filter out the bug reports and commits that do not reference the word test. This step removes most irrelevant bug reports and commits that do not deal with a flaky test directly. Finally, we manually inspect the remaining bug reports/commits to filter out false positives – bug reports/commits that actually are not related to a flaky test/fix but still match our keyword search. We inspect the bug reports first, because they usually contain a good description of the flaky test and a possible fix. On the other hand, commit messages often leave out details concerning why or how flaky tests are flaky. In some cases, the bug reports also have related fix commits for flaky tests. Afterwards, when inspecting the commits, we ignore the commits already referenced by the bug reports.

Table 2 shows the breakdown of our filtering process. We find 75 bug reports/commits related to fixing a flaky test across 20 projects.

## 2.3 Analyzing the Bug Reports and Commits

We divide the filtered bug-reports and commits among the authors of the paper. For each test fixed in a bug report or commit, we aim to classify the cause for the flaky test and the type of fix for the flaky test. First, an author independently reasoned in detail about each assigned bug to determine categories for the cause and fix. After that, another author double-checks each bug-report/commit for any incorrect classifications. Finally, we discuss together to determine the distinct categories and merge all the results. We describe these categories next.

## 2.4 Causes of Flaky Tests

From our manual inspection, we create eight categories for the causes of the flaky tests in our study. Table 3 shows the breakdowns.

<sup>1</sup>We use the dependent “packages” as reported by the GitHub API, which are projects that can compile into libraries to be used by others. We use packages because they are more likely to be actively maintained by developers and have reasonable test suites.

**Table 3: Causes of Flakiness**

CauseCategory	# of Bug Reports/Commits
Algorithmic Non-determinism	45
Floating-point Computations	5
Incorrect/Flaky API Usage	4
Unsynced Seeds	2
Concurrency	2
Hardware	1
Other	12
Unknown	4
<b>Total</b>	<b>75</b>

**2.4.1 Algorithmic Non-determinism.** We classify a bug report/commit in this category when the cause of flakiness is due to inherent non-determinism in the algorithm being used in the test. Probabilistic programming systems and machine learning frameworks provide functionality for computations that are inherently non-deterministic. For instance, Bayesian inference algorithms like Markov Chain Monte Carlo (MCMC) [44] compute the posterior distribution of a given statistical model by conditioning on observed data. MCMC guarantees convergence to the target posterior distribution in the limit. When designing a test, the developer typically chooses a simple model and a small dataset, which ideally converges very fast. The developer then adds an assertion checking whether the inferred parameter (mean) is close to the expected result. However, there is always a non-zero probability that the algorithm may not converge even with the same number of iterations. Hence, developers need to choose a suitable assertion that accounts for this randomness but does not let any bugs slip through. This behavior is also common for other systems using deep learning algorithms and natural language processing. Given all the parameters a developer has to consider when designing such tests, it is not surprising that the majority of flaky tests in this domain are due to this reason. There are five bug reports/commits related to the randomness of input data, three due to non-determinism in model sampling, and 37 due to non-determinism of algorithms during the training process. Out of those 37 bug reports/commits related to non-determinism of training algorithms, 14 of them involve NLP training algorithms [6], 12 involve deep learning algorithms [61], six involve deep reinforcement learning algorithms [23], and five involve weak supervision training algorithms [72].

**Example.** The *rlworkgroup/garage* project provides a toolkit for developing and evaluating Reinforcement Learning (RL) algorithms. It also includes implementations of some RL algorithms. Listing 1a shows an example of a test (simplified) for the DQN (Deep Q-Network) model [43], which is used for learning to control agents using high-dimensional inputs (like images). First, the test chooses model training parameters (Lines 21–25). Second, it initializes a game environment (Line 26), which the model should learn to play and chooses an exploration strategy (Line 27). Third, it initializes the DQN algorithm with several parameters, including the exploration strategy and number of training steps (Lines 30–31). Fourth, it trains the model using specified training parameters and returns the average score (`last_avg_ret`) obtained by the model (Lines 32–35). Finally, the assertion checks whether the average score is greater than 20 (Line 58). The algorithm is, however, non-deterministic in

nature – at any step of the game (when the method `get_action` in Listing 1b is called), the algorithm chooses a random action (shown in Listing 1b - Line 64–65) with some probability. Hence, the score obtained by the algorithm varies across runs (even for same parameters), and it sometimes is less than the asserted 20, causing the test to occasionally fail, i.e., it is a flaky test. In Section 2.5.1, we discuss how developers fix such flaky tests.

---

```
# test_dqn.py
18 def test_dqn_cartpole(self):
19     """Test DQN with CartPole environment."""
20     with LocalRunner(self.sess) as runner:
21         n_epochs = 10
22         n_epoch_cycles = 10
23         sampler_batch_size = 500
24         num_timesteps = n_epochs * n_epoch_cycles *
25             sampler_batch_size
26         env = TfEnv(gym.make('CartPole-v0'))
27         epsilon_greedy_strategy = EpsilonGreedyStrategy(
28             env_spec=env.spec,
29             total_timesteps=num_timesteps, ...)
30         algo = DQN(env_spec=env.spec,
31             exploration_strategy=epsilon_greedy_strategy, ...)
32         runner.setup(algo, env)
33         last_avg_ret = runner.train(
34             n_epochs=n_epochs,
35             n_epoch_cycles=n_epoch_cycles,
36             batch_size=sampler_batch_size)
37         assert last_avg_ret > 20
```

---

(a) Flaky test in `test_dqn.py`


---

```
# epsilon_greedy_strategy.py
61 def get_action(self, t, observation, policy, **kwargs):
62     opt_action = policy.get_action(observation)
63     self._decay()
64     if np.random.random() < self._epsilon:
65         opt_action = self._action_space.sample()
66     return opt_action, dict()
```

---

(b) Source of randomness in `get_action`

### Listing 1: A Flaky Test caused due to Algorithmic Non-Determinism

**2.4.2 Floating-point Computations.** We classify a bug report/commit in this category if the test is flaky due to incorrect handling of floating-point computations such as not handling special values (such as NaN) or having rounding issues. However, the floating point computations only result in these erroneous conditions for certain sequences of values, which can differ across executions due to randomness. Hence, tests sporadically fail due to incorrect handling of these special values.

**2.4.3 Incorrect/Flaky API Usage.** We classify a bug report/commit in this category if the related code uses an API incorrectly, or if the API is known to be flaky but is not handled appropriately in source/test code. As an example, in the *rlworkgroup/garage* project, there are two tests that are testing some functionality of the project that involves training a TensorFlow computation graph (which persists across tests). However, neither of the tests reset the graph

after use. Hence, when run back to back, TensorFlow crashes due to duplicate variables in the graph, so the test that runs after fails.

**2.4.4 Unsynced Seeds.** We classify a bug report/commit in this category if the test is setting seeds for random number generators inconsistently. Many of the libraries that we study use multiple modules that rely on a notion of randomness. For example, *tensorflow/tensor2tensor* uses tensorflow, numpy, and python’s `random` module. Using different seeds (or not setting seeds) across these systems can trigger different computations, therefore leading to different test results than expected.

**2.4.5 Concurrency.** We classify a bug report/commit in this category when the flakiness is caused due to interaction among multiple threads. Different runs of the same test leads to different thread inter-leavings, which in turn lead to different test results. As an example, in the *tensorflow/tensor2tensor* project, there was a control dependency on an input for a computation in the RNN. However, when the tests run in parallel, the order of computations is uncertain, which causes some tests to fail when the input has not yet been computed.

**2.4.6 Hardware.** We classify a bug report/commit in this category if the flakiness is from running on some specialized hardware. We find *one commit* that disables a test on TPU (Tensor Processing Unit), which is a specialized accelerator for deep learning. TPUs are efficient in doing parallel computations like matrix multiplications on a very large scale. However, these computations can be relatively non-deterministic due to different orderings of floating point computations. The randomness in the order of collecting partial results from several threads can sometimes amplify the errors, leading to different results.

**2.4.7 Other.** We group several bug reports/commits that are likely flaky due to causes related to flaky tests in general software into this category. These flaky tests include test failures due to flakiness in the pylint checker, file system, network delays, and iterating over an unordered key set in a python dictionary. There are 12 bug reports/commits in this category.

**2.4.8 Unknown.** We classify a bug report/commit into the Unknown category when we do not find enough information to properly categorize the cause for flakiness. These bug reports/issues do not provide enough description of the cause of flakiness, neither in the commit message nor in the developer conversations.

## 2.5 Fixes for Flaky Tests

Table 4 shows the common categories of fixes we observe for flaky tests in our evaluation projects. Note that the total number differs by four from the total in Table 3 (79 versus 75), because four of the bug reports/commits includes two fixes each that we classify into two separate categories.

**2.5.1 Adjust Thresholds.** In many cases, the developers reason that the threshold for the assertion in a test is too tight, causing the test to fail intermittently. In such scenarios, the developers prefer loosening the threshold by some amount to reduce the test flakiness.

One example test is `test_mnist_tutorial_tf` in the *tensorflow/cleverhans* project. The test runs various ML algorithms on

the MNIST dataset, perturbed by adversarial attacks. It then checks various accuracy values generated in the report. Originally, the test’s assertion compares the computed value with 0.06.

However, the developers observed that, on adversarial examples, the accuracy occasionally exceeds 0.06. To fix [12] this flaky test, the developers used a higher threshold of 0.07.

```
self.assertTrue(report.clean_train_adv_eval < 0.06 )
self.assertTrue(report.clean_train_adv_eval < 0.07 )
```

**2.5.2 Fix Test and Fix Code.** We classify 13 bug reports/commits where developers fix a bug in the *test code* and 9 bug reports/commits where developers fix a bug in *source code*, to mitigate flakiness completely in the test instead of reducing the chances of flaky failure (e.g., by adjusting thresholds in the assertion).

These two categories each cast a fairly wide net, as there is no *one* kind of fix in this category that is common across all the projects. For instance, Issue #727 of the *allenai/allennlp* project [3] exposes the issue that their tests were creating modules during execution, but python’s `importlib` could not find them. To fix this, developers added `importlib.invalidate_caches()` (as recommended by the `importlib` documentation) to the *test code* so that the newly created modules can be discovered. In another case, the loss computation in the *cornellius-gp/gpytorch* project was buggy since it was not handling NaNs gracefully, leading to intermittent failures during execution. In Pull Request #373 [31], the developers added a check in the *source code* to account for NaNs during execution and provide warning messages to the user instead of crashing.

**2.5.3 Fix Seed.** Fixing a seed for a non-deterministic algorithm makes the computations deterministic and easier for the developers to write assertions that compare against a fixed value.

An example from Pull Request #1399 in the project *PySyft* [55] illustrates this strategy. Originally, the developers had a test called `test_federated_learning` that creates a dataset and a neural network model, and finally performs a few iterations of back-propagation. Then, the test checks whether the loss in the last iteration is less than the loss in the first iteration.

**Table 4: Fixes for Flaky Tests**

Fix Category	# of Bug Reports/Commits
Adjust Thresholds	15
Fix Test	13
Fix Seed	12
Remove Test	10
Mark Flaky Test	10
Fix Code	9
Adjust Test Params	8
Upgrade Dependencies	1
Other	1
<b>Total</b>	<b>79</b>

The total number differs by four from the total in Table 3 (79 versus 75), because four of the bug reports/commits include two fixes each that we classify into two separate categories.

```
1 for iter in range(6):
2     for iter in range(2):
3         for data, target in datasets:
4             model.send(data.owners[0])
5             model.zero_grad()
6             pred = model(data)
7             loss = ((pred - target)**2).sum()
8             loss.backward()
9             ...
10            if(iter == 0):
11                first_loss_loss = loss.get().data[0]
12            assert loss.get().data[0] < first_loss
13            if(iter == 1):
14                final_loss = loss.get().data[0]
15            assert final_loss == 0.18085284531116486
```

This test was problematic due to non-deterministic floating point computations of gradients. As a result, there is always a possibility that the loss may not reduce in a small number of steps (6 in this case). To resolve the flakiness, the developers instead fixed the seed earlier in the code using `torch.manual_seed(42)`, changed the test to perform just *two* iterations of back-propagation, and finally changed the assertion to check the *exact* value of the loss.

This test, while seemingly fragile, is now deterministic due to the fixed seed. In general, fixing seeds can make the test deterministic. However, the test can potentially fail in the future if the sequence of underlying computations (which deal with randomness) changes due to modifications of the code under test or if the implementation of the random number generator changes. We provide a more detailed qualitative discussion on the trade-offs of setting seeds in the test code through a few case studies in Section 5.

**2.5.4 Remove Test.** Developers may remove or disable a flaky test if they cannot fix the flakiness or tune the various parameters to control the results. While such a solution does indeed “fix” a flaky test, it is a rather extreme measure as a developer is deciding that the flakiness in the test is more trouble than the benefit the test provides. However, we still found 10 fixes where the developer removes or disables a flaky test.

**2.5.5 Mark Flaky Test.** The *Flaky* plugin [22] for pytest allows the developers to mark a test as flaky. The developer can annotate a test function with the `@flaky` decorator, and then pytest by default re-runs the test once in the case of a failure. The developer can also set the maximum number of re-tries (`max_runs`) and minimum number of passes (`min_passes`). pytest re-runs the test until it passes `min_passes` times or runs the test `max_runs` of times. The developers usually prefer this setup when the test fails in some corner cases that are expected to occur rarely but hard to handle specifically.

For example, in the *allenai/allennlp* project, the developers added the `@flaky` decorator to `test_model_can_train_save_and_load` [2], forcing pytest to rerun it in case of future failures.

**2.5.6 Adjust Test Params.** The accuracy of a machine learning or Bayesian inference algorithm depends on the number of iterations/epochs on which the model is trained. If the number of iterations is not enough, then the results may not be stable and can occasionally cause the assertions in tests, which check for accuracy,

**Table 5: Distribution of Fixes for Each Cause of Flakiness**

Cause\ Fix	Adjust Thresholds	Fix Test	Fix Seed	Remove Test	Mark Flaky Test	Fix Code	Adjust Test Params	Upgrade Deps	Other	Total
Algorithmic Non-det	14	2	11	5	9	0	8	0	0	49
FP Computations	1	1	0	0	0	3	0	0	0	5
Incorrect/Flaky API Usage	0	2	0	0	0	2	0	0	0	4
Unsynced Seeds	0	0	1	1	0	0	0	0	0	2
Concurrency	0	1	0	0	0	1	0	0	0	2
Hardware	0	0	0	1	0	0	0	0	0	1
Other	0	6	0	1	1	3	0	1	0	12
Unknown	0	1	0	2	0	0	0	0	1	4
<b>Total</b>	<b>15</b>	<b>13</b>	<b>12</b>	<b>10</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>1</b>	<b>1</b>	<b>79</b>

to fail. Apart from number of iterations/epochs, there are other test parameters that a user can configure, like batch size, data set generation parameters (size and distribution), and number of reruns (for known flaky tests).

For example, in the *allenai/allennlp* project, developers noted that the *NumericallyAugmentedQaNetTest*, from *naqanet\_test.py*, was flaky and can cause failures when run on GPU. So, in commit 089d744 [1], the developers increased the maximum number of runs and specified the minimum number of passes needed for the test to be marked successful:

```
@flaky (max_runs=3, min_passes=1)
def test_model_can_train_save_and_load(self):
    self.ensure_model_can_train_save_and_load(self.param_file)
```

The `max_runs` and `min_passes` parameters are part of the *Flaky* plugin [22].

**2.5.7 Upgrade Dependencies.** Developers sometimes have to upgrade their existing dependency versions, such as a dependency on a probabilistic programming system or machine learning framework. We observed 1 bug report from the project *azavea/raster-vision*, where developers upgraded the version of TensorFlow on which the project depends.

In Issue #285 in the *azavea/raster-vision* project [57], the developers were observing an intermittent failure that occurred roughly ten percent of the time. The failure was triggered by using the command `export_inference_graph` from the TensorFlow Object Detection API, which would intermittently give a `Bad file descriptor` error. The developers resolved this issue through upgrading their dependency on TensorFlow 1.5 to TensorFlow 1.8.

**2.5.8 Other.** In Issue #167 of the *tensorflow/cleverhans* project [13], a developer reported that some tests were both slow to run and flaky. The developers applied a number of fixes to update the test configurations and updated various test parameters to improve the runtime of the tests. We classified this fix as “Other” as it did not directly fit into any other category.

## 2.6 Distribution of Fixes for Each Cause

Table 5 presents the distribution of fixes for each cause of flakiness. For the biggest cause of flakiness, Algorithmic Non-determinism, we see that the most common developer fixes are to adjust the

accuracy thresholds in assertions, fix seeds, mark the tests as flaky (equivalent to re-running), or even remove the test. This analysis brings out two main observations: (1) Fixing flaky tests in the Algorithmic Non-determinism cause category requires an intimate understanding of the code under test and hence is non-trivial, sometimes even for developers familiar with the code base. (2) We observe that adjusting accuracy thresholds in assertions is the most common fix developers choose overall.

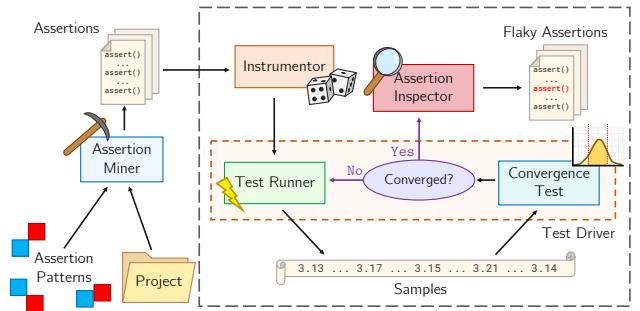
These two observations motivated us to develop FLASH, a novel technique for detecting flaky tests caused by Algorithmic Non-determinism that focuses on tests with assertions that do approximate comparisons (using developer-specified accuracy thresholds) between actual and expected values in tests.

## 3 FLASH

We propose FLASH, a technique for detecting flaky tests due to non-deterministic behavior of algorithms. At a high-level, FLASH runs tests multiple times but with different random number seeds, which leads to different sequences of random numbers between executions. FLASH then checks for differences in actual values in test assertions and even different test outcomes. The main challenge to this approach is to determine how many times FLASH should run each test before it can make a decision as to whether the test is flaky. We address this challenge by using a *convergence test* (Section 3.2) to dynamically determine how many times to run a test.

### 3.1 System Overview

Figure 2 shows the overall architecture for FLASH. FLASH takes as input a project and a set of assertion patterns to check for flakiness.

**Figure 2: Flash Architecture**

FLASH reports the assertions in the project that match the input assertion patterns and that can fail due to sampling random numbers. FLASH consists of four main components: (1) the Assertion Miner collects all the assertions that match the assertion patterns specified by the user, (2) the Test Instrumentor instruments a test assertion to set necessary seeds and log the actual and expected values in the assertion, (3) the Test Driver runs the test several times until the distribution of actual values converge, and (4) the Assertion Inspector compares the actual and expected values and determines the probability that the assertion might fail. Algorithm 1 describes the main algorithm for FLASH and how it uses all four components. While FLASH currently runs on all tests in a project to detect flaky tests (and we run on all for our later evaluation), developers can instead run FLASH for tests only after they fail, or check only newly committed tests, speeding up the flaky test detection over time.

Next, we first discuss the details of the convergence test, and then we discuss each of FLASH's components in detail and how they use the convergence test.

---

**Algorithm 1** FLASH Algorithm
 

---

```

Input: Project  $P$ , Assertion patterns  $A_p$ 
Output: Set of resulting flaky assertions  $FA$ 

1: procedure FLASH( $P, A_p$ )
2:    $FA \leftarrow \emptyset$ 
3:    $As \leftarrow AssertionMiner(P, A_p)$ 
4:   for  $(T, A) \in As$  do
5:      $T_i \leftarrow TestInstrumentor(T, A)$ 
6:      $S \leftarrow TestDriver(T_i)$ 
7:      $status, \mathcal{P} \leftarrow AssertionInspector(T, A, S) = FLAKY$ 
8:     if  $status = FLAKY$  then
9:        $FA \leftarrow FA \cup \{(A, S, \mathcal{P})\}$ 
10:    end if
11:   end for
12: return  $FA$ 
13: end procedure
  
```

---

### 3.2 Convergence Test

Given an assertion  $A$  in a test function  $T$ , we want to compute the probability of failure for the assertion. Computing this probability requires the entire distribution of values that the expression in the assertion can evaluate to. Let us assume we have an assertion of the following form:

$$\text{assert } x < \theta$$

Here, we would like to estimate the distribution for  $x$  so that we can compute the probability of  $x$  exceeding  $\theta$ .

We pose this problem in the form of estimating the distribution of an unknown function  $\mathcal{F}$ , where  $\mathcal{F}$  essentially runs  $T$  and returns the value of  $x$ . One approach for solving this problem is to use a sampling-based approach, wherein we execute  $\mathcal{F}$  multiple times to obtain a number of samples, estimate the distribution from the samples, and compute the probability of failure :  $P(\mathcal{F} > \theta)$ . However, this approach has two main challenges. We need to decide (1) whether *we have seen enough samples* and (2) *how many samples* to collect at minimum.

To tackle the first challenge, researchers have proposed several metrics to measure convergence of a distribution like Gelman-Rubin diagnostic [26], Geweke diagnostic [27], and Raftery-Lewis [56]. In this work, we specifically use the Geweke diagnostic [27] to measure the convergence of a set of samples. Intuitively, the Geweke

diagnostic checks whether the mean of the first 10% of the samples is not significantly different from the last 50%. If yes, then we can say that the distribution has converged to the target distribution. To measure the difference between the two parts, the Geweke diagnostic computes the Z-score, which is computed as the difference between the two sample means divided by the standard errors. Equation 1 shows the formula for the Z-score computation for Geweke diagnostic, where  $a$  is the early interval of samples,  $b$  is the later interval of samples,  $\hat{\lambda}$  is the mean of each interval and  $Var$  is the variance of each interval of samples.

$$z = \frac{\hat{\lambda}_a - \hat{\lambda}_b}{\sqrt{Var(\lambda_a) + Var(\lambda_b)}} \quad (1)$$

If the Z-score is small, then we can say that the distribution has converged. We choose to use the Geweke diagnostic because it does not involve any assumptions about the independence of the samples and does not require samples from multiple chains like other metrics. To use this metric with our approach, the user can specify the minimum desired threshold. In FLASH, we use a threshold of 1.0 in the algorithm, which intuitively translates to choosing a maximum standard deviation of 1.0 between intervals.

Unfortunately, there is no one way to tackle the second challenge i.e., *how many samples* to collect at minimum. If the minimum sample size is too large, we waste too much time executing  $\mathcal{F}$  even when not needed. On the other hand, if the sample size is too small, the conclusions may be questionable. Some studies [58] recommend using a minimum sample size of 30 for statistical significance. We allow the user to specify the minimum sample size, but by default, and for our own evaluation (Section 4), we set it to 30 samples.

**Computing probability of failure.** Given a set of samples for  $\mathcal{F}$ , we now need to determine the probability of failure:  $P(\mathcal{F} > \theta)$ . For this task, we fit an empirical distribution,  $\mathcal{D}$ , over the samples, and compute the cumulative distribution function (CDF):  $CDF_{\mathcal{D}}(\theta) = P(\mathcal{D} < \theta)$ . Finally, the probability of failure is given by  $1 - P(\mathcal{D} < \theta)$ . We can easily transform any other kind of assertion into this form to do this computation as well.

**Comparison with hypothesis testing.** An alternate approach to using a convergence test is to use statistical hypothesis testing. In this case, we would try to reason about two hypotheses: the null hypothesis:  $P(\mathcal{F} > \theta) > k$ , and the alternative hypothesis:  $P(\mathcal{F} > \theta) \leq k$ . A common hypothesis test is the sequential probability ratio test (SPRT) [68], which continuously samples from the given function until it either rejects the null hypothesis or accepts the alternative hypothesis. SPRT is proven to take the optimal number of iterations for a desired level of accuracy. For this approach, we can model  $\mathcal{F}$  as a binomial random variable and only record whether it passed or failed after each execution. Each sample of  $\mathcal{F}$  is assumed to be independent.

However, there are several practical limitations of using hypothesis testing for our purposes. First, to obtain a desired level of accuracy, one needs to collect a very large number of samples. For instance, for  $k = 0.01$ , to obtain a Type I error of less than 0.01 and Type II error of less than 0.2, one needs at least 100 samples. For a non-flaky assertion where the values of  $x$  are not non-deterministic, hypothesis testing would still require many samples to determine the assertion is not flaky, wasting time. Second, a flaky assertion

**Table 6: Assertion Patterns**

Source	Assertion	Count
Python	assert expr <   >   <=   >= threshold	62
Unittest	assertTrue	64
Unittest	assertFalse	0
Unittest	assertGreater	168
Unittest	assertGreaterEqual	7
Unittest	assertLess	467
Unittest	assertLessEqual	21
Numpy	assert_almost_equal	252
Numpy	assert_approx_equal	8
Numpy	assert_array_almost_equal	222
Numpy	assert_allclose	278
Numpy	assert_array_less	4
TensorFlow	assertAllClose	1613
Total		3166

may exhibit a large variation in actual observations but may not fail in the first 100 iterations (due to pure chance). The SPRT test would miss detecting this flaky test. The convergence test, however, has a better chance of capturing this behaviour as it monitors the variation in the overall distribution of values. Given these limitations, we choose to use a convergence test to determine the number of times to run a test for our technique FLASH.

### 3.3 FLASH Components

We now describe the main components of FLASH.

**3.3.1 Assertion Miner.** We consider only assertion methods that do approximate comparisons between the actual computed value and the expected value rather than checking for exact equality. These assertions are similar to approximation oracles as defined by Nejadgholi and Yang in their prior work [45]. Table 6 shows the assertion patterns that we consider along with what library the assertion comes from and how often each assertion appears within our evaluation projects.

In Algorithm 1, FLASH calls the Assertion Miner to get all the assertions that match the specified assertion patterns in the project (line 3). The Assertion Miner iterates through all the test files in a project and checks each test function to see if it contains one or more assertions of interest. The Assertion Miner creates an *assertion record* for each assertion of interest, which consists of the file name, class name, test function name, and assertion location in the file.

**3.3.2 Test Instrumentor.** For each assertion record, FLASH uses the Test Instrumentor to instrument the relevant assertion for the later steps that runs the test (line 5 of Algorithm 1). The instrumentation involves logging the actual and expected values during a test run and introducing the logic for controlling the seed for the random number generators during the test run.

The Test Instrumentor sets the seed by introducing a pytest fixture function that performs setup before tests run. The fixture sets a concrete seed for the random number generator in different modules (e.g., NumPy, TensorFlow, PyTorch). We configure the pytest fixture to run at the session level, which means that the fixture runs once before any test runs. If the developer has explicitly set some seed for their tests, the fixture would not override that seed.

**3.3.3 Test Driver.** FLASH takes the instrumented test and passes it to the Test Driver to run and collect samples for the relevant assertion (line 6 in Algorithm 1). The Test Driver relies on a convergence

**Algorithm 2** Test Driver Algorithm

```

Input: Instrumented test  $T_i$ 
Output: Set of samples  $S$ 

1: procedure TESTDRIVER( $T_i$ )
2:    $batch\_size = INITIAL\_BATCH\_SIZE$ 
3:    $i \leftarrow 0$ 
4:    $S \leftarrow \emptyset$ 
5:   while  $i < MAX\_ITERS$  do
6:      $b \leftarrow 0$ 
7:     while  $b < batch\_size$  do
8:        $sample \leftarrow ExecuteTest(T_i)$ 
9:        $S \leftarrow S \cup \{sample\}$ 
10:       $b \leftarrow b + 1$ 
11:    end while
12:     $score \leftarrow ConvergenceScore(samples)$ 
13:    if  $score < CONV\_THRESHOLD$  then
14:      break
15:    end if
16:     $i \leftarrow i + batch\_size$ 
17:     $batch\_size \leftarrow BATCH\_UPDATE\_SIZE$ 
18:  end while
19:  return  $S$ 
20: end procedure

```

test to determine how many samples it needs to collect (which is the number of times to run the test).

Algorithm 2 shows the high-level algorithm for the Test Driver. Aside from the input instrumented test  $T_i$ , the Test Driver requires the user to set up some other configuration options related to the convergence test (Section 3.2). These options include the initial number of iterations to run the test ( $INITIAL\_BATCH\_SIZE$ , default 30), the maximum iterations to run if the values do not converge ( $MAX\_ITERS$ , default 500), the maximum threshold for the convergence test score ( $CONV\_THRESHOLD$ , default 1.0), and the additional number of iterations to run at a time if the samples do not converge ( $BATCH\_UPDATE\_SIZE$ , default 10). The Test Driver then returns the set of collected samples.

TestDriver runs the test multiple times using the *ExecuteTest* procedure (line 8 in Algorithm 2). The *ExecuteTest* procedure runs the test using pytest in a virtual environment setup for the project. After running the test, *ExecuteTest* returns a *sample*, which consists of the actual value from the assertion, the expected value from the assertion, a log of the output from the test run, and the seed set to the random number generators for that one run. The actual and expected value can be either a scalar, array, or tensor. Furthermore, *ExecuteTest* can obtain multiple actual and expected values for a single assertion, if the assertion is called multiple times within a single test run, e.g., the assertion is in a loop. In such cases, the actual and expected values in the sample are arrays containing the values from each time the assertion is executed in the test run.

If the convergence test finds the samples to have converged after the initial batch of runs (lines 7–12), then the Test Driver continues to collect more samples by iterating in batches of  $BATCH\_UPDATE\_SIZE$  iterations (line 17). The Test Driver returns the set of collected samples after convergence or after reaching  $MAX\_ITERS$  iterations.

**3.3.4 Assertion Inspector.** FLASH uses the Assertion Inspector to determine if an assertion is flaky or not (line 7 in Algorithm 1). The Assertion Inspector first checks if any of the collected samples contain a failure (the assertion was passing in the production environment before running with FLASH, so a failure should indicate flakiness). If not, the Assertion Inspector takes the samples and the

assertion, and it computes the probability of the assertion failing (Section 3.2). If the probability of failure is above the user-specified threshold, the assertion is also considered flaky. FLASH records all the assertions the Assertion Inspector reports as flaky, along with collected samples and probability of failure  $\mathcal{P}$  for such assertions.

The Assertion Inspector also reports the bounds of the distribution of actual values sampled. The user can use the reports from the Assertion Inspector to make a decision on whether to take action in fixing the assertion or not, along with information that can help with fixing the assertion.

## 4 EVALUATION

We evaluate FLASH on the same projects where we find flaky tests from historical bug reports/commits from Section 2. Specifically, we answer the following research questions:

- RQ1** How many new flaky tests does FLASH detect?
- RQ2** How do developers react to flaky tests FLASH detects?
- RQ3** How many old, historical flaky tests does FLASH detect?

**Experimental Setup.** We run all our experiments using a 3 GHz Intel Xeon machine with 12 Cores and 64GB RAM. We implement FLASH entirely using Python. We use the Geweke test implementation provided by the Arviz library [35] in Python.

### 4.1 RQ1: New Flaky Tests Detected by FLASH

For the 20 projects from our study with flaky test bug reports/commits, we run FLASH on each project’s latest commit, collected on Dec 20, 2019. For each project, we create a virtual environment using the Anaconda package management system [14] for Python to run the tests, installing the latest version of each project’s specified dependencies. We use a threshold of 1.0 for convergence test using Geweke diagnostic (CONV\_THRESHOLD). INITIAL\_BATCH\_SIZE is set to 30 and BATCH\_UPDATE\_SIZE is set to 10. We are not able to run FLASH for three projects. Of these three projects, we do not find assertions matching the assertion patterns we support in FLASH for *pytorch/captum* and *azavea/raster-vision*. For the remaining project, *rlworkgroup/garage*, the tests time out when run with FLASH due to the limited GPU support that we have on our machine, which causes those tests to run very slowly.

Table 7 shows our results for running FLASH on the remaining projects. The first column **Projects** shows the name of the project. The second column **Total** shows the total number of assertions FLASH collects samples from for each project. The third column **Pass** shows the number of assertions that always pass across all runs. The fourth column **Fail** shows the number of assertions that fail at least once when FLASH runs them for several iterations. The fifth column **Skip** shows the number of assertions that were skipped due to several reasons (e.g., needing specialized hardware like a TPU). The sixth column **Conv-NZ** shows the assertions that have a convergence score of more than 0 in the first batch. The seventh column **Conv-Z** shows the number of assertions that have a 0 convergence score in the first batch. The eighth column **Avg-Runs** shows the average number of iterations that FLASH runs for the assertions. The last column **Max-Runs** shows the maximum number of iterations that FLASH runs for the assertions.

From the **Conv-Z** column, we see that most of the assertions always have the same actual value, which is why the convergence

score is 0 in these cases. As such, the average number of runs is usually quite close to the initial batch size we set (30). The average number of iterations per assertion over all projects is 45.32 (and the average of the maximum number of iterations per assertion is 233.53, suggesting the convergence test is effective at reducing the number of iterations FLASH should run per assertion).

There are 252 cases where an assertion never fails but has a non-zero convergence score. The values of the expressions in these assertions fluctuate, but we do not observe any failure. We rerun FLASH for those assertions with a stricter threshold of 0.5 for the convergence test and observe whether they reveal any more failures. On average, while this configuration option makes FLASH take more iterations for convergence, we still do not observe any failures.

Table 8 shows the details of the failing tests that FLASH detects (we do not find a case where FLASH finds multiple assertions in the same test to fail). The second column lists the total failing tests for each project. We manually inspect each failure and reason about whether the failure indicates a potential flaky test in the project under test. The third column shows the number of tests we determine to be a new flaky test. The fourth column shows the number of tests where the developers confirm the flaky test after we report them. The final column shows the number of tests where we determine that the cause of failure is either due to a mis-configuration in our local test environment or it is an already confirmed flaky test. Overall, FLASH detects 32 failing assertions across 11 projects, of which we determine 11 indicate flaky tests.

### 4.2 RQ2: Fixing Flaky Tests

For the 11 potential flaky tests FLASH detects, we sent 4 Pull Requests (PRs) and 6 Issue Requests to the developers of the projects on GitHub. Developers accepted 3 PRs that fix four flaky tests, while the other is closed. For the closed PR (in *zfit/zfit*), the developers confirmed that the test is flaky but made a different fix than what we proposed. Out of the 6 Issue Requests, the developers have confirmed 5 and fixed one of them, and the last is pending review. We discuss the flaky tests that are confirmed and fixed in more detail.

FLASH finds four flaky tests in *allenai/allennlp*. In one flaky test, the test randomly generates a list of predictions and corresponding labels, then computes the AUC (Area Under Curve) metric and checks whether the result matches the expected result, which is computed using a similar implementation from the *scikit-learn* package. However, the metric fails when there are no positive labels in the randomly generated list of labels. We could easily reproduce the error using the seed(s) FLASH finds. After discussing with the developers, we conclude that this is an expected behaviour for the AUC implementation. Hence, we fix the test by explicitly setting one label to positive always before calling the metric. In the three other flaky tests, the tests run various training algorithms and compute their F1 scores (harmonic mean of precision and recall). Then, the tests check whether the computed F1 score is greater than 0. FLASH finds at least one PyTorch seed for each test that causes the F1 score to be 0, and hence the assertion fails. After discussing this issue with the developers, we conclude that the best fix is to mark the test as flaky. An F1 score of 0 is expected when there are no positive samples in a dataset.

**Table 7: Distribution of Passing and Failing Asserts**

<b>Projects</b>	<b>Total</b>	<b>Pass</b>	<b>Fail</b>	<b>Skip</b>	<b>Conv-NZ</b>	<b>Conv-Z</b>	<b>Avg-Runs</b>	<b>Max-Runs</b>
allenai/allennlp	428	424	4	0	97	331	43.29	200
cornellius-gp/gpytorch	1180	1179	1	0	126	1054	35.66	200
deepmind/sonnet	185	166	1	18	0	167	30.00	30
geomstats/geomstats	467	464	3	0	0	467	30.00	30
HazyResearch/metal	19	18	1	0	3	16	35.79	80
OpenMined/PySyft	12	11	1	0	8	4	110.83	500
PrincetonUniversity/PsyNeuLink	198	166	0	32	0	166	30.00	30
pytorch/botorch	86	85	1	0	2	84	35.81	500
pytorch/vision	7	7	0	0	3	4	42.86	80
Qiskit/qiskit-aqua	113	75	9	29	7	77	53.33	500
RasaHQ/rasa	19	19	0	0	9	10	85.26	400
snorkel-team/snorkel	58	58	0	0	7	51	42.76	500
tensorflow/cleverhans	79	75	2	2	6	71	31.82	70
tensorflow/gan	108	100	0	8	2	98	31.00	90
tensorflow/magenta	21	21	0	0	0	21	30.00	30
tensorflow/tensor2tensor	157	150	7	0	9	148	38.22	500
zfit/zfit	29	27	2	0	13	16	63.79	230
<b>Sum/Avg</b>	<b>3166</b>	<b>3045</b>	<b>32</b>	<b>89</b>	<b>292</b>	<b>2785</b>	<b>45.32</b>	<b>233.53</b>

**Table 8: Failures for Each Project**

Project	Failures	Flaky Tests	Confirmed	Other
allenai/allennlp	4	4	4	0
cornellius-gp/gpytorch	1	0	0	1
deepmind/sonnet	1	1	1	0
geomstats/geomstats	3	0	0	3
HazyResearch/metal	1	0	0	1
OpenMined/PySyft	1	1	1	0
pytorch/botorch	1	1	1	0
Qiskit/qiskit-aqua	9	0	0	9
tensorflow/cleverhans	2	2	2	0
tensorflow/tensor2tensor	7	1	0	6
zfit/zfit	2	1	1	1
<b>Total</b>	<b>32</b>	<b>11</b>	<b>10</b>	<b>21</b>

FLASH finds one new flaky test in *zfit/zfit*. The test creates four random variables, each of which is assigned a Gaussian distribution. Then, it creates another random variable that is just the sum over those four Gaussian random variables. Finally, the test fetches the dependent random variables from that summed random variable through the call `get_dependents()`, which returns an *ordered set*. The test asserts that the iteration order of the ordered set is different than the iteration order of the regular, unordered set of these dependent random variables. FLASH finds an execution where this test fails, which is expected since there is always a non-zero probability that iterating through the unordered set leads to the same order as the ordered set. We proposed a fix to this test in a PR, where we marked it as flaky using the `@flaky` annotation in pytest. The developers confirmed that the test is flaky (and buggy) but they made a different fix. They reasoned that the test should just be checking that multiple calls to `get_dependents()` return the same ordered set, so they refactored the test to check that the return of `get_dependents()` is the same across subsequent calls.

**Table 9: Old Flaky Tests Detected by FLASH**

Project	Commit	Cause	Failures/Iters	Threshold
allenai/allennlp	5e68d04	FP Computations	2/100	1.0
allenai/allennlp	5dd1997	Algo. Non-det.	28/100	1.0
tensorflow/tensor2tensor	6edb6b	Algo. Non-det.	2/70	1.0
pytorch/botorch	e2e132d	Algo. Non-det.	2/170	0.5
pytorch/botorch	7ac0273	Algo. Non-det.	6/50	1.0
tensorflow/cleverhans	b2bb73a	Algo. Non-det.	5/40	1.0
tensorflow/cleverhans	4249afc	Algo. Non-det.	19/80	1.0
tensorflow/cleverhans	58505ce	Algo. Non-det.	7/50	1.0
snorkel-team/snorkel	3d8ca08	Algo. Non-det.	1/90	0.5
OpenMined/PySyft	b221776	Algo. Non-det.	1/270	0.5
pytorch/captum	d44db43	Algo. Non-det.	4/40	1.0

FLASH finds another flaky test in *pytorch/botorch*. The test creates a probabilistic model, generates data, runs a few steps of gradient computation, and checks whether the gradient is greater than a fixed threshold. However, FLASH finds an execution and the corresponding Torch seed where this checks fails. After reporting this behavior to the developers, they confirm that the test is indeed flaky and adjust the threshold to reduce the chance of failure.

### 4.3 RQ3: Old Flaky Tests Detected by FLASH

For the same 20 projects, we evaluate how effective FLASH is at detecting the old, already identified flaky tests. We only run FLASH for flaky tests that fail due to the assertions FLASH tracks.

First, for each commit related to a fix for a flaky test from our study, we checkout its immediate parent commit (before the fix) and create a virtual environment to run the test on that commit. Like in Section 4.1, we create this virtual environment using Anaconda, but we ensure that the version of each dependency we install is set to the latest version at the time of the commit, not the current latest, as to better reproduce the environment developers were initially running the tests and encountering flakiness. Once we set up this

environment, we run FLASH only on the flaky test in question to see if FLASH can detect this flaky test. We use the same configuration for FLASH as we describe in Section 4.1.

Table 9 shows our results on using FLASH to detect old flaky tests. The table shows the project and old commit SHA where we run FLASH, the cause of the flaky test, the fix for the flaky test by the developers, the number of iterations FLASH runs the test based from using Algorithm 1, the number of times the test fails in those iterations, and the threshold we eventually set for the Geweke test to check for convergence. We are unable to run FLASH on the other old flaky tests, because either we cannot reproduce the exact environment to make any tests pass on the old commit, or the old flaky test in question was not failing due to assertions FLASH supports. For example, there are projects where we could not reproduce the environment due missing old dependencies.

Overall, FLASH requires at most 100 iterations to detect most of these flaky tests (only two require more iterations). For three flaky tests, we have to use a tighter threshold of 0.5 to detect the known flaky test. A tighter threshold is expected in some cases, because it might require longer to converge to the target distribution.

## 5 DISCUSSION

For many assertions, FLASH obtains the same actual value for every run, despite changing the seeds. We suspect that developers are purposely setting the seed to a specific value tests, overriding FLASH’s attempt at changing the seed. Setting the seed guarantees a deterministic execution of the tests, a common fix for flaky tests (Section 2.5). However, setting the seed to a specific value can “lock” the developer into a specific execution for all test runs, and developers can potentially miss bugs if other seeds lead to different values that the code is not handling properly. Furthermore, if the underlying random number generator changes, or if the developer changes their code where only the random number generator’s sequence of random numbers changes, the assertions can end up failing, because they are too dependent on the specific sequence of random numbers and are therefore flaky.

To evaluate how flaky existing assertions are when run under different seeds for a project, we disable all explicit seed settings throughout the tests in the project<sup>2</sup>. Then, we run FLASH on this modified code. We experiment on two projects: *PrincetonUniversity/PsyNeuLink* and *geomstats/geomstats*.

We sample several of the assertions that FLASH finds as flaky after running on the modified code. For the assertions we determine to be flaky with our manual inspection, we send PR(s) to the developers for fixing the flakiness. We observe both a positive response and a negative response from the two projects.

**Experience with *geomstats/geomstats*.** Tests in *geomstats/geomstats* run with different backends, including NumPy, TensorFlow, and PyTorch. We run FLASH on the project using each backend while changing seeds. FLASH finds only one test in *geomstats/geomstats* that fails when run with the TensorFlow backend, but always passes in other backends. The test creates an object that represents an N-dimensional space, samples a random point from the defined space, then checks whether the sample belongs to the

<sup>2</sup>Interestingly, developers across different projects like to use very similar seeds, such as 0, 1234, or 42.

defined space using the exposed API. The test uses a predefined tolerance of  $1e^{-6}$  for the check. After reporting to the developers, they confirmed that this is likely a precision issue: the TensorFlow backend uses single precision floating point by default, whereas the NumPy backend uses double precision. Hence, the NumPy backend can handle higher tolerance levels than TensorFlow for this test. We send a PR to reduce the tolerance level to  $1e^{-4}$ , which the developers accepted. This case demonstrates that setting seeds in tests can be problematic and sometimes hide subtle bugs. Hence, developers must be careful about using fixed seeds in their tests and reason judiciously about the entailing risks of fixed seeds.

**Experience with *PrincetonUniversity/PsyNeuLink*.** FLASH finds several tests in *PrincetonUniversity/PsyNeuLink* to fail when run with different seeds. We send a PR to fix one test to start. The test is testing an integrator mechanism that gives deterministic results modulo noise sampled from a normal distribution from NumPy. The assertion checks that the actual value should be close to the expected value with a very specific tolerance level, which is the actual value from running with the specific seed set by the developers. We also find that developers had to several times update the assertion in the past due to changes in the sequence of random numbers, even though they never change the seed itself. We update the assertion to accept a wider tolerance, representing the distribution of values FLASH reports. The goal is to make the assertion fail less often in the future due to changes in just the sequence of random numbers.

However, the developers were against the change, because they indeed want the test to fail even when the failure is solely due to changes in the sequence of random numbers. They want to examine *all* failures, and if they decide the problem is not in their code, they will update the expected value of the assertion accordingly. As such, the developers seem not bothered by the times the test has a flaky failure unrelated to changes in their code under test.

## 6 THREATS TO VALIDITY

We study only a subset of projects that use probabilistic programming systems and machine learning frameworks, so our results may not generalize to all projects. To mitigate this threat, we study four popular open-source probabilistic programming systems and machine learning frameworks on GitHub, and we consider all of their dependent projects with more than 10 stars (indicating they are somewhat popular and used). Our study on historical bug reports/commits referencing flaky tests may not include all such bug reports/commits. We use a keyword search similar to prior work on studying flaky tests in project histories. We are still able to identify a substantial number of bug reports/commits referencing flaky tests.

FLASH samples seeds to use for random number generators, so the test executions we run through FLASH is only a subset of all possible test executions given different possible sequences of random numbers. The tests FLASH finds to be flaky are for sure flaky, as we confirm the test can pass when run on the code, and we can reproduce the test failure using the seeds FLASH reports. Furthermore, we report these flaky tests to developers and receive confirmation from them that we have detected flaky tests. There may be more flaky tests that FLASH does not detect due to not running enough. As such, our results on flaky tests FLASH detects is an under-count of how many flaky tests exist in these projects.

## 7 RELATED WORK

**Testing of systems dealing with randomness.** Machine learning frameworks like TensorFlow [64] and PyTorch [51] have revolutionized the domain of machine learning. A recent surge of interest in probabilistic programming has also led to the development of numerous probabilistic programming systems both in academic research and industry [10, 11, 28–30, 41, 42, 46, 52, 54, 67, 69]. However, tests written by developers in these domains suffer from the problem of inherent non-determinism and lack concrete oracles. Recent work has proposed techniques to systematically test and debug probabilistic programming systems [18, 19], machine learning frameworks [20, 53], and randomized algorithms [34].

Nejadgholi and Yang [45] studied the distribution and nature of approximate assertions in deep learning libraries. They report *adjusting thresholds* as one class of changes that developers typically do for approximate assertions. We also find fixes in the same category in our study (Section 2.5), and we made similar fixes in Section 4.2. However, in other cases in our study, the fixes are more involved and belong to other categories.

**Study of Flaky Tests.** Recently, there has been much work on studying flaky tests. Harman and O’Hearn [32] reported the problems with flaky tests at Facebook, and they have even suggested that future testing research should adjust to assuming all tests to be flaky. Luo et al. studied the various causes and fixes for flaky tests in open-source software [40]. They studied flaky tests in traditional software, finding that common causes for flaky tests include async wait, concurrency, and test-order dependencies. In this work, we study flaky tests specifically in the domain of software that depends on probabilistic programming systems and machine learning frameworks. In our study, we find that most causes of flaky tests would fall under the category prior work would consider as “randomness”, which did not show up as a prominent cause of flakiness in their evaluation. Zhang et al. conducted a survey of machine learning testing and found that flaky tests arise in metamorphic testing whenever floating point computations are involved [71]. In our study, we find floating-point computations to be a major cause for flakiness, such as tests not handling special values (such as NaN) or having rounding issues.

**Detecting Flaky Tests.** Concerning flaky test detection, Bell et al. proposed DeFlaker [8], a technique for detecting when test failures after a change are due to flaky tests by comparing the coverage of the failing tests with the changed code. There has also been work on techniques that detect specific types of flaky tests. Lam et al. proposed iDFlakies [38], a framework for detecting order-dependent flaky tests, tests that fail when run in different orders, by running tests in different orders. Gambi et al. also detect order-dependent flaky tests using PRADET [24], a technique that tracks data dependencies between tests. Shi et al. proposed NonDex [62], a technique for detecting flaky tests that assume deterministic iteration order over unordered collections by randomly shuffling unordered collections and checking if tests fail. Our tool FLASH also focuses on a specific type of flaky tests that rely on random number generators used through probabilistic programming system and machine learning frameworks. FLASH accomplishes this task through running tests multiple times under different seeds, while using statistical tests to determine the number of runs.

There has also been prior work on analyzing the impact of flakiness. Cordy et al. proposed FlakiMe [15], a laboratory-controlled test flakiness impact assessment and experimentation platform, that supports the seeding of a (controllable) degree of flakiness into the behavior of a given test suite. Our tool FLASH also supports setting seeds for non-deterministic flaky tests and checking the posterior distribution of the testing objects.

**Debugging and Fixing Flaky Tests.** Recent work has started to focus on how to debug and ultimately fix flaky tests. Lam et al. developed a framework at Microsoft to instrument flaky test executions and collect logs of the traces for both passing and failing executions, which they then find differences between so they can determine the root cause of flakiness [36]. Lam et al. followed up this work with an additional study on how developers at Microsoft attempt to fix flaky tests [37]. Based on their study, they proposed a technique for handling the common case of async waits by modifying wait times in the async waits to reduce flakiness [37]. Shi et al. studied how to fix another type of flaky tests, order-dependent flaky tests, finding that order-dependent flaky tests can be fixed using code taken from other tests in the test suite [63].

**Fuzz Testing.** Our approach for FLASH is also similar to fuzz testing. Fuzz testing involves generating random data as inputs to software to find bugs in the software. Fuzz testing has been used to find security vulnerabilities [4], performance hot-spots [39], bugs in probabilistic programming systems [18], etc. Many recent fuzz testing techniques are guided to increase coverage of code as to find more bugs [48–50]. While fuzz testing techniques generate random values that are explicitly passed as inputs to the code under test, FLASH generates random values for the seeds to random number generators that code depends on, exploring how different seed values can lead to different test outcomes when existing tests are run on the same version of code.

## 8 CONCLUSION

Randomness is an important trait of many probabilistic programming and machine learning applications. At the same time, software developers who write and maintain these applications often do not have adequate intuition and tools for testing these applications, resulting in flaky tests and brittle software. In this paper, we conduct the first study of flaky tests in projects that use probabilistic programming systems and machine learning frameworks. Our investigation of 75 bug reports/commits that reference flaky tests in 20 projects identified Algorithmic Non-determinism to be a major cause of flaky tests. We also observe that developers commonly fix such flaky tests by adjusting assertion thresholds. Inspired by the results of our study, we propose FLASH, a technique for systematically running tests with different random number generator seeds to detect flaky tests. FLASH detects 11 new flaky tests that we report to developers, with 10 already confirmed and 6 fixed. We believe that a new generation of software testing tools (like FLASH) based on the foundations of the theory of probability and statistics is necessary to improve the reliability of emerging applications.

## ACKNOWLEDGMENTS

This work was partially funded by NSF grants CNS-1646305, CCF-1703637, OAC-1839010, and CCF-1846354.

## REFERENCES

- [1] AllenNLP Commit 089d744 2019. <https://github.com/allenai/allennlp/pull/2778/commits/089d744>.
- [2] AllenNLP Commit 53bba3d 2018. <https://github.com/allenai/allennlp/commit/53bba3d>.
- [3] AllenNLP Issue 727 2018. <https://github.com/allenai/allennlp/pull/727>.
- [4] American Fuzzy Loop 2014. <http://lcamtuf.coredump.cx/afl>.
- [5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* (2015).
- [6] M. Bates. 1995. Models of natural language understanding. *Proceedings of the National Academy of Sciences of the United States of America* (1995).
- [7] Matthew James Beal. 2003. *Variational algorithms for approximate Bayesian inference*.
- [8] Jonathan Bell, Owolabi Legunse, Michael Hilton, Lamya Eloussi, Tiffany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*.
- [9] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* (2019).
- [10] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. 2016. Stan: A probabilistic programming language. *JSTATSOFT* (2016).
- [11] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *ESEC/FSE*.
- [12] Cleverhans Commit 58505ce 2017. <https://github.com/tensorflow/cleverhans/pull/149/commits/58505ce>.
- [13] Cleverhans Issue 167 2017. <https://github.com/tensorflow/cleverhans/issues/167>.
- [14] Conda package management system 2017. <https://docs.conda.io>.
- [15] Maxime Cordy, Renaud Rwmelika, Mike Papadakis, and Mark Harman. 2019. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment: A Case Study on Mutation Testing and Program Repair. *arXiv:1912.03197 [cs.SE]*
- [16] Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Vikash K. Mansinghka, and Martin Vechev. 2018. Incremental Inference for Probabilistic Programs. In *PLDI*.
- [17] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv preprint arXiv:1711.10604* (2017).
- [18] Saikat Dutta, Owolabi Legunse, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *ESEC/FSE*.
- [19] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *ESEC/FSE*.
- [20] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghavam M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*.
- [21] Eric Jang. Why Randomness is Important for Deep Learning 2016. <https://blog.evjang.com/2016/07/randomness-deep-learning.html>.
- [22] Flaky test plugin 2019. <https://github.com/box/flaky>.
- [23] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. 2018. An Introduction to Deep Reinforcement Learning. *arXiv:1811.12560 [cs.LG]*
- [24] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *ICST*.
- [25] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.
- [26] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis*.
- [27] John Geweke et al. 1991. *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments*. Federal Reserve Bank of Minneapolis, Research Department Minneapolis, MN.
- [28] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* (1994).
- [29] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).
- [30] Noah D Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages. <http://dippl.org>.
- [31] GPyTorch Pull Request 373 2018. <https://github.com/cornelliou-gp/gpytorch/pull/373>.
- [32] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *SCAM*.
- [33] Jason Brownlee. Embrace Randomness in Machine Learning 2019. <https://machinelearningmastery.com/randomness-in-machine-learning/>.
- [34] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical algorithmic profiling for randomized approximate programs. In *ICSE*.
- [35] Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo A. Martin. 2019. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *The Journal of Open Source Software* (2019).
- [36] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *ISSTA*.
- [37] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *ICSE*.
- [38] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakes: A Framework for Detecting and Partially Classifying Flaky Tests. In *ICST*.
- [39] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *ISSTA*.
- [40] Qingzhou Luo, Farah Hariri, Lamya Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
- [41] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099* (2014).
- [42] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yang, A. Spangler, and J. Bronskill. 2013. Infer.NET 2.5. Microsoft Research Cambridge. <http://research.microsoft.com/infernet>.
- [43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
- [44] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo* (2011).
- [45] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *ASE*.
- [46] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.
- [47] Akira K Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* (1998).
- [48] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *ISSTA DEMO*.
- [49] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *ISSTA*.
- [50] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang. OOPSLA* (2019).
- [51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*.
- [52] Avi Pfeffer. 2001. IBAL: a probabilistic rational programming language. In *IJCAI*.
- [53] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*.
- [54] PyroWebPage 2018. Pyro. <http://pyro.ai>.
- [55] PySyft Issue 1399 2018. <https://github.com/OpenMined/PySyft/pull/1399>.
- [56] Adrian E Raftery and Steven M Lewis. 1995. The number of iterations, convergence diagnostics and generic Metropolis algorithms. *Practical Markov Chain Monte Carlo* (1995).
- [57] Raster Vision Issue 285 2018. <https://github.com/azavea/raster-vision/issues/285>.
- [58] John A Rice. 2006. *Mathematical statistics and data analysis*.
- [59] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* (2016).
- [60] Simone Scardapane and Dianhui Wang. 2017. Randomness in neural networks: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (2017).
- [61] Jurgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* (2015).
- [62] August Shi, Alex Gyori, Owolabi Legunse, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST*.
- [63] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakes: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*.
- [64] TensorFlowWebPage 2018. TensorFlow. <https://www.tensorflow.org>.
- [65] Swapna Thorve, Chandani Sreshta, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *ICSME*.
- [66] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *ICLR*.
- [67] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv* (2016).
- [68] Abraham Wald. 1945. Sequential tests of statistical hypotheses. *The annals of mathematical statistics* (1945).

- [69] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *AISTATS*.
- [70] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification, and Reliability* (2012).
- [71] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. arXiv:[1906.10742](https://arxiv.org/abs/1906.10742) [cs.LG]
- [72] Zhi-Hua Zhou. 2017. A Brief Introduction to Weakly Supervised Learning. *National Science Review* (2017).

# Scaffold: Bug Localization on Millions of Files

Michael Pradel\*  
University of Stuttgart  
Germany

Mateusz Machalica  
Facebook  
USA

Vijayaraghavan Murali  
Facebook  
USA

Erik Meijer  
Facebook  
USA

Rebecca Qian  
Facebook  
USA

Satish Chandra  
Facebook  
USA

## ABSTRACT

Despite all efforts to avoid bugs, software sometimes crashes in the field, leaving crash traces as the only information to localize the problem. Prior approaches on localizing where to fix the root cause of a crash do not scale well to ultra-large scale, heterogeneous code bases that contain millions of code files written in multiple programming languages. This paper presents Scaffold, the first scalable bug localization technique, which is based on the key insight to divide the problem into two easier sub-problems. First, a trained machine learning model predicts which lines of a raw crash trace are most informative for localizing the bug. Then, these lines are fed to an information retrieval-based search engine to retrieve file paths in the code base, predicting which file to change to address the crash. The approach does not make any assumptions about the format of a crash trace or the language that produces it. We evaluate Scaffold with tens of thousands of crash traces produced by a large-scale industrial code base at Facebook that contains millions of possible bug locations and that powers tools used by billions of people. The results show that the approach correctly predicts the file to fix for 40% to 60% (50% to 70%) of all crash traces within the top-1 (top-5) predictions. Moreover, Scaffold improves over several baseline approaches, including an existing classification-based approach, a scalable variant of existing information retrieval-based approaches, and a set of hand-tuned, industrially deployed heuristics.

## CCS CONCEPTS

- Software and its engineering → Software maintenance tools; Software creation and management;

## KEYWORDS

Bug localization, software crashes, machine learning

### ACM Reference Format:

Michael Pradel, Vijayaraghavan Murali, Rebecca Qian, Mateusz Machalica, Erik Meijer, and Satish Chandra. 2020. Scaffold: Bug Localization on Millions of Files. In *Proceedings of the 29th ACM SIGSOFT International Symposium on*

\*Work mostly performed while on sabbatical at Facebook, Menlo Park.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397356>

*Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.*  
ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397356>

## 1 INTRODUCTION

When software crashes in the field, often the only information available to developers are *crash traces*. Such traces come in various forms and may include various kinds of hints about the code that causes the crash, including stack traces, error messages, information about the program state just before the crash, and additional information added by tools that process crash traces. To prevent future instances of the same crash, developers must identify where exactly in the code base to fix the underlying problem – a problem referred to as *bug localization*.

### 1.1 Crash-Based Bug Localization at Scale

We focus on localizing bugs at the file-level within ultra-large scale, heterogeneous code bases, i.e., code bases that contain multiple millions of code files written in multiple programming languages. Addressing the file-level bug localization problem at scale can help with the process of handling field crashes in multiple ways. By pinpointing which files are most relevant for a crash, bug localization helps find the right team or person to address the crash. Finding the right developer for a given crash is non-trivial in a large organization, and associating crashes to the wrong developers consumes valuable time and resources. Once the right developer has been identified, knowing the buggy file helps the developer to focus on where to implement a fix of a crash. Moreover, identifying the file to fix can serve as a first step in automated program repair [14].

Unfortunately, manual bug localization is difficult and time-consuming. One reason is that crash traces contain lots of noise not directly related to the bug location. Another reason is that, for very large-scale code bases, there may be millions of code files to choose from. The problem is further compounded by the fact that widely deployed software may lead to such a volume of crashes per day that is practically impossible to process without appropriate tools.

### 1.2 Scalability Problems of Prior Work

Prior work has proposed several automatic bug localization techniques, which are extremely inspiring, but not easily applicable to ultra-large scale, heterogeneous code bases, such as those that motivate this work. Broadly speaking, one group of prior work is based on traces of correct and buggy executions [1, 6, 15]. These approaches assume that a test suite is available where some tests pass while other tests fail due to the bug. Unfortunately, failing tests

**Table 1: Prior work on bug localization.**

Work	Evidence of bug	Technique	Lang.	Code analysis	Scalability issue
[18]	Bug report	Topic modeling	Java	Yes	Extracts topics from each source file.
[33]	Stack trace	IR	Java	Yes	Extracts natural language words from each source file.
[21]	Bug report	IR	Java	Yes	Parses and analyzes each source file.
[25]	Set of stack traces	Clustering & learning to rank	Java, C/C++	No	Queries a learned model for each pair of crash and source file.
[9]	Bug report	Classifier	C/C++	No	Classifier where each source file is a separate class.
[30]	Set of stack traces	Static analysis & heuristics	C/C++	Yes	Call graph analysis, control flow analysis, and slicing for each source file.
[32]	Bug report	Learning to rank	Java	Yes	Extracts natural language words from each source file.
[27]	Bug report	IR	Java	Yes	Analyzes and segments each source file.
[17]	Bug report	Static analysis & IR trace	Java	Yes	Analysis of control flow and data flow dependencies.
[13]	Bug report	Learning & IR	Java	Yes	Extracts features from each source file.
[19]	Bug report	IR	Java	Yes	Extracts terms from each source file.
[12]	Bug report	Multiple classifiers	Java	Yes	Analyzes all source files; queries learned models for each pair of bug and source file.
This work	Raw crash traces	Learning & IR	Lang.-indep.	No	None

that reproduce field crashes often are not available, and generating such tests is a challenging problem on its own [22].

The other group of prior work is based on some evidence of the bug, such as a stack trace produced by the bug or a natural language bug report. Table 1 summarizes the most closely related approaches in this group. Many existing approaches statically analyze each file in the code base, e.g., to analyze dependencies between code elements or to extract features of individual files [12, 13, 17–19, 21, 27, 30, 32, 33]. The extracted information can then, e.g., be fed into an information retrieval (IR) component that compares the words in a bug report or stack trace with the tokens in code files [13, 17, 19, 21, 27, 33]. Another direction addresses bug localization as a learning problem, e.g., via a classification model that considers each file in the code base as a separate class [9].

The main limitation of these previous approaches is the *lack of scalability* to ultra-large scale, heterogeneous code bases. Analyzing each source file in a code base that contains millions of files written in multiple languages is far from trivial, even with a lightweight analysis. None of such approaches [12, 13, 17–19, 21, 27, 30, 32, 33] has been applied at the scale targeted here. To the best of our knowledge, the largest code base used in prior work [12] consists of 71,000 Java files from 45 projects, i.e., two orders of magnitude smaller

than the multi-million code base considered here. The approaches that do not require any source file analysis suffer from their own scalability issues. Work that considers each source file as a separate class for a classifier [9] does not scale well, as we show experimentally in Section 6.5. Other work, which queries a trained model for each pair of a crash and a source file [25], takes at least a linear (w.r.t. the number of source files) amount of time for each individual crash.

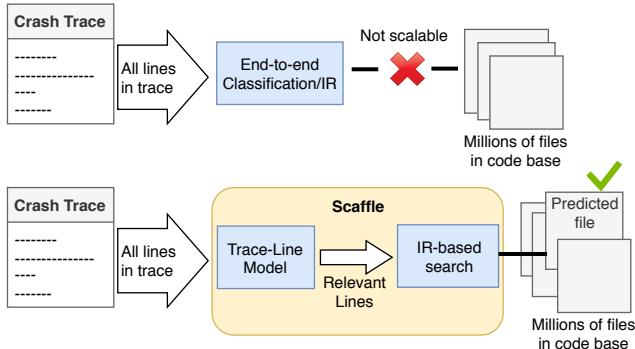
Moreover, almost all existing techniques (except [25]) target stack traces produced by a single programming language, typically Java, building on parsers, regular expressions, and heuristics specialized for stack traces in this language. Unfortunately, a single-language approach is difficult to adopt to crash traces that originate from several programming languages, come in various different formats, and may have been processed by a diverse and evolving set of tools.

### 1.3 Our Work in a Nutshell

This paper presents Scaffle, the first bug localization technique for crashes caused by code in ultra-large scale, heterogeneous code bases. To scale Scaffle to code bases with multi-million files, and tens of thousands of crashes, we use the key insight that *the problem of localizing bugs based on crash traces can be decomposed into two sub-problems*. Figure 1 shows a high-level overview of this decomposition. Existing bug localization techniques (top) address the problem through an end-to-end approach that directly compares the crash trace to files in the code base. Instead, Scaffle (bottom) decomposes the problem into two sub-problems, each addressed by a separate component.

The first component of Scaffle addresses the problem of identifying the most relevant lines of a given crash trace. This component, called the *trace-line model*, is a machine learning model that reads all lines of the given trace and assigns a relevance score to each line. We implement the trace-line model using neural network-based supervised learning, which learns from past crashes and the bug fix locations associated with them. The second component of Scaffle addresses the problem of matching the most relevant lines in a crash trace with file paths in the code base. We view this problem as an IR problem where a line of a crash trace serves as a query over the file paths in the code base. Decomposing the problem is inspired by the observation that a single line in a crash trace often provides most hints to localize the buggy file, while the other lines add various kinds of noise.

Our work breaks with two assumptions made by prior work. First, the approach does not assume that one can analyze the entire code base. Instead, Scaffle matches specific lines in a crash trace with paths in the code base, without ever analyzing the content of the files stored at these paths. In particular, Scaffle avoids statically analyzing all files in the code base. Second, the approach does not assume a specific programming language, and as a result, also does not assume a specific structure or format of crash traces. Instead of building, fine-tuning, and constantly evolving specialized parsers for different trace formats, the model automatically learns to parse and understand a diverse set of crash traces. As a result of these design decisions, the approach scales well to very large code bases that contain code written in multiple languages.



**Figure 1: Overview of Scaffle (bottom) and end-to-end methods (top).**

We evaluate our work by applying it to tens of thousands of crash traces produced by a large-scale code base at Facebook, which contains several millions of files. To the best of our knowledge, our dataset is an order of magnitude larger than previous evaluations of crash trace-based bug localization techniques, and we apply Scaffle to a code base that is at least two orders of magnitude larger than in any prior work [12].

The empirical results show that decomposing the bug localization problem into two sub-problems is key to obtaining an approach that is effective for a large-scale code base. We find that the model is effective at predicting those lines in a trace that pinpoint the buggy file, reaching a mean reciprocal rank of over 0.8. Using this model, Scaffle correctly predicts a to-be-fixed file for 40% to 60% (50% to 70%) of all crash traces in the top-1 (top-5) predictions. The effectiveness is roughly the same across different programming languages and parts of the code base.

Comparing Scaffle to prior work highlights that end-to-end classification [9] does not scale well to millions of files, and that a baseline (relying on no source analysis) end-to-end IR-based bug localization in the vein of prior work [13, 17, 19, 21, 27, 33] is less effective than our approach.

In summary, this paper makes the following contributions:

- A scalable, language-independent technique to predict from raw crash traces which files to change to address a crash.
- The insight that the crash-based bug localization problem can be decomposed into two simpler problems: identifying relevant lines in a crash trace and matching those lines with file paths in the code base.
- A learned model for predicting the most relevant lines of crash traces. The model is realized as a neural network that is trained in a supervised manner on past fix locations.
- Empirical evidence from applying our approach in a large-scale, industrial setting, showing that Scaffle effectively predicts those parts of the code base to focus on to fix a crash.

## 2 OVERVIEW AND EXAMPLE

Before describing the details of Scaffle in Sections 3 and 4, this section illustrates the main ideas with an example. The inputs given to Scaffle are a *raw crash trace* and the set of all file paths in the code base. By raw crash trace, we mean any kind of structured or unstructured text that we assume to be separated into lines. In

particular, these traces may contain one or more stack traces, information about the application and the underlying system where the crash occurred, and information about the state of the application. Figures 2(a) and (b) give two examples of crash traces, one produced by a crash in Java code and one produced by a crash in PHP code. We assume a single representative crash trace as the input to Scaffle, e.g., identified by techniques for clustering and prioritizing crashes [3–5, 8, 10, 23]. The file paths in the code base each are a sequence of path segments, e.g., “proj/pkg/someFile.java”.

The first component of Scaffle, the trace-line model, identifies those lines of a crash trace that are most relevant for localizing the bug. To obtain this component, we exploit the insight that most large-scale projects have plenty of historical data about crashes and about code changes that fix the root cause of crashes. Given the fix location of a crash, we derive how relevant each line of the crash trace is for finding the bug location, based on whether the line contains parts of the file path of the bug location. Scaffle then learns from this data a machine learning model that summarizes individual lines of a crash trace and then predicts the relevance of each of these lines (Section 3). Learning a model, instead of hard-coding a set of heuristics, addresses the challenge that the format of crash traces not only varies, but also evolves over time, as new programming languages and APIs become popular.

For the examples in Figure 2(a) and (b), the right-most column shows the relevance score that our neural trace-line model predicts for each line. Intuitively, the lines with highest relevance contain the most information about file paths likely to be changed to fix the bug. As illustrated by Figure 2(a), some of the most relevant lines may be among the first lines of a crash trace, e.g., because it summarizes the location where an exception was thrown. As illustrated by Figure 2(b), the most relevant lines may also be somewhere in the middle of a crash trace. Section 3 describes in detail the trace-line model that predicts which of the given lines are most relevant.

The second component of Scaffle is an IR-based search for the most likely bug locations. In IR, a query usually needs to be matched against a possibly large set of documents. In our case, one of the most relevant lines of the crash trace is the query, and each file path in the code base is a document. Specifically, we tokenize all file paths into segments, and similarly tokenize the words appearing in the most relevant line. We then feed it as the query into an IR-based search engine, and return the file paths from the search results as a ranked list of possibly buggy files. The underlying assumption is that the most relevant trace line often contains some path segments of the buggy file, even though it may not refer to the exact path.

For our running example, line 28 of the Java trace in Figure 2(a), i.e., the most relevant line, will be tokenized into the keywords “at dostuff dostuffbrowsercontroller exitdomorestuff dostuffbrowsercontroller java” for the query. For this query, a path “projectX/packageA/dostuff/DoStuffBrowserController.java” would be considered more similar to the query than a path “projectX/packageB/some/other/path/Logger.java”, and hence will be predicted as the most likely path. Section 4 describes our approach for matching trace lines with file paths in detail.

(a) Example of a Java crash trace.

Nb.	Lines of raw crash trace relevance	Predicted
1	android_crash:java.lang.IllegalStateException:dostuff.DoStuffBrowserController.clearBrowserFragment	83%
2	stack_trace: java.lang.RuntimeException: Unable to destroy activity {lala/lala.LoginActivity}: ...	58%
3	at android.app.ActivityThread.performDestroyActivity(ActivityThread.java:3975)	14%
...	(dozens of more lines)	
23	Caused by: java.lang.IllegalStateException: Activity has been destroyed	12%
24	at android.app.FragmentManagerImpl.enqueueAction(FragmentManager.java:1376)	21%
25	at android.app.BackStackRecord.commitInternal(BackStackRecord.java:745)	12%
26	at android.app.BackStackRecord.commitAllowingStateLoss(BackStackRecord.java:725)	30%
27	at dostuff.DoStuffBrowserController.clearBrowserFragment(DoStuffBrowserController.java:775)	85%
28	at dostuff.DoStuffBrowserController.exitDoMoreStuff(DoStuffBrowserController.java:196)	100%
29	at domorestuff.DoMoreStuffRootView.exitDoMoreStuff(DoMoreStuffRootView.java:554)	87%
...	(dozens of more lines)	
76	app_upgrade_time: 2018-08-19T17:02:12.000+08:00	5%
77	package_name: lala	1%
78	peak_memory_heap_allocation: 92094532	1%
79	app_backgrounded: false	13%
...	(dozens of more lines)	

(b) Example of a PHP crash trace.

Nb.	Lines of raw crash trace relevance	Predicted
1	[twi01447.08.ftw1.example.com] [Sat Sep 29 13:20:40 2018] ...	9%
2	(Events: <null_response_query_id>)	2%
3	(App Version: 189.0.0.44.93)	0%
4	(NNTraceID: FAiHFWy4Isj)	1%
5	(Sampling ID: A_oSj-oZbo1GJnM8JHKL3o_)	1%
...	(dozens of other lines)	
31	trace starts at [/var/abc/def/core/runtime/error.php:1021]	35%
32	#0 __log_helper(...) called at [/var/abc/def/core/runtime/error.php:1021]	47%
33	#1 log() called at [/var/abc/def/core/logger/logger.php:1162]	49%
34	#2 NNDefaultLogMessage->log() called at [/var/abc/def/core/logger/logger.php:938]	41%
35	#3 NNLogMessage->process() called at [/var/abc/def/core/logger/logger.php:804]	46%
36	#4 NNLogMessage->warn() called at [/var/abc/def/core/logger/logger.php:791]	65%
37	#5 NNLogMessage->warn_High() called at [/var/abc/def/foo/multifoo/client/base/MultifooClient.php:1518]	82%
38	#6 MultifooClient->genPopulateResults\$memozie_Impl() called at [/var/abc/def/foo/multifoo/client/base/MultifooClient.php:2248]	82%
39	#7 MultifooClient->genQueryID() called at [/var/abc/def/bar/query/multifoo/MultifooQuery.php:5782]	77%
40	#8 Closure\$MultifooQuery::genStuff#9() called at [/var/abc/def/gates/core/Gate.php:390]	68%
...	(dozens of other lines)	

Figure 2: Examples of Scaffle’s approach to crash-based bug localization (Note: The traces and file paths are made-up but modeled after real data.)

### 3 PREDICTING RELEVANT LINES IN RAW TRACES

This section describes Scaffle’s approach for predicting the most relevant lines in a given crash trace, which is the first of two steps for predicting the bug location. Our approach is based on the observation that for many crash traces, a single line is sufficient to pinpoint the bug location, while dozens or even hundreds of other lines are irrelevant. The reason may be, e.g., that the bug location is on the stack when the crashes happens, and hence occurs in a single frame of the stack trace, or that the crash trace mentions a specific term that matches the buggy path.

Scaffle trains a machine learning model to identify the most relevant lines in a trace. The motivation for choosing a learning-based approach over, e.g., hard-coding heuristics for a specific trace format, is two-fold. First, learning from data allows the approach to cover traces produced by multiple programming languages and coming in different formats. Second, re-training the model with recent data allows for easily adapting the approach to evolving crash traces, e.g., when the popularity of programming languages and APIs changes over time.

The model addresses the following problem:

*Definition 3.1 (Line prediction problem).* Given a crash trace  $t = (l_1, \dots, l_n)$  that consists of  $n$  lines, predict a vector of relevance scores  $r = (s_1, \dots, s_n)$ , such that lines with a higher relevance score contain more information about the bug location.

**Algorithm 1** Gathering training data for the trace-line model.**Input:** Trace  $t$ , changeset  $C$ **Output:** Vector  $r$  of relevance scores

```

 $r \leftarrow$  new vector
for all line  $l$  in  $t$  do
     $s_{max} \leftarrow 0$ 
     $T \leftarrow tokenize(l)$ 
    for all path  $p$  in  $C$  do
         $P \leftarrow tokenize(p)$ 
         $P \cap T$ 
         $s \leftarrow \frac{P \cap T}{P}$ 
        if  $s > s_{max}$  then
             $s_{max} = s$ 
    Append  $s_{max}$  to  $r$ 

```

The remainder of this section presents how to gather data to train a supervised model that addresses the above problem (Section 3.1) and a neural network architecture we use to learn the model (Section 3.2).

### 3.1 Obtaining Historical Training Data

We address the line prediction problem through supervised machine learning, i.e., by learning from crash traces labeled with their most relevant lines. Scaffold creates this data by gathering pairs of crashes and fixes in the history of the code base and by computing a projected ground truth from these pairs. Each data point in the resulting training data is a pair  $(t, r)$  of a crash trace  $t$  and its corresponding vector  $r$  of relevance scores. Intuitively,  $r$  assigns those lines the highest relevance that point to (or that at least resemble) the paths in the code base where the crash was fixed.

To obtain crash-relevance pairs  $(t, r)$ , we use crash traces associated with code changes that fix the root cause of the crash. For gathering crash traces and their associated changesets at Facebook, we perform an approach similar to past work that extracted such data from open-source repositories [30]. Some crashes are associated with issues that track progress toward fixing the underlying bug. Once fixed, the issue refers to the changeset that implements the fix. We gather pairs of crashes and changesets by combining both associations.

Given a pair  $(t, c)$  of a crash trace  $t$  and a changeset  $c$ , Scaffold obtains a crash-relevance pair  $(t, r)$  by comparing the paths affected by  $c$  with the lines in  $t$ . Algorithm 1 summarizes this step. It performs a pairwise comparison of each line in the trace and each file path affected by the changeset, which yields the relevance of the line for predicting the path. For this comparison, Scaffold tokenizes file paths into individual path segments and lines into individual words. Our tokenization function for trace lines does not assume any particular structure, but simply splits lines at every non-alphabetic character. Given a tokenized file path and a tokenized line, the algorithm computes the percentage of words in the path that are also contained in the line. Finally, the best score of a line across all file paths in the changeset is added to the relevance vector.

For illustration, reconsider the example in Figure 2(a) and suppose that it is part of the data that the trace-line model is learned from. Line 27 is tokenized into a sequence of words (“at”, “dostuff”, “DoStuffBrowserController”, “clearBrowserFragment”, “java”, “775”).

Suppose that the changeset consists of a single path “projectX/packageA/dostuff/DoStuffBrowserController.java”, which we tokenize into (“projectX”, “packageA”, “dostuff”, “DoStuffBrowserController”, “java”). Because line 27 shares 3 out of 5 words with the path, the relevance score of the line is set to  $\frac{3}{5} = 60\%$ . Other lines contain less relevant information. For example, line 24 shares only the word “java” with the path, and therefore it is assigned a relevance score of  $\frac{1}{5} = 20\%$ . After computing the relevance score of each line, the algorithm concatenates all scores into a single relevance vector.

The algorithm for gathering labeled training data is a simple heuristic to identify the most relevant lines of a trace. In principle, the word-based matching of lines against file paths may not find the optimal relevance scores, and alternative definitions of Algorithm 1 exist. In practice, we find the algorithm to be an efficient way of producing training data that yields an effective trace-line model.

### 3.2 Neural Trace-Line Model

Scaffold uses the historical data, extracted as described above, as the ground truth for training a model that predicts the relevance of each line in a raw crash trace. In principle, different kinds of models could be trained for this purpose. We here present a neural network-based model, since neural networks have been proven to be highly effective at reasoning about raw input data, without the need to define and extract features of the input. The model reasons about a trace as a sequence of lines, each of which is a sequence of words. To tokenize a trace, we use the same tokenizer in Algorithm 1, which splits the text at every non-alphabetic character.

Figure 3 gives an overview of the neural network architecture. The network consists of two bi-directional recurrent neural networks (RNNs). The first RNN, called the *line-level RNN*, summarizes the words of each line into a continuous vector representation. The second RNN, called the *trace-level RNN*, takes a sequence of line-level vectors and predicts the relevance score of each line. Intuitively, this decomposition reflects how a human understands a trace, i.e., by understanding individual lines and by then reasoning about the meaning of multiple lines.

The input given to the trace-line model is a sequence of lines  $t = (l_1, \dots, l_n)$ , where each line consists of a sequence of words  $l_i = (w_1, \dots, w_k)$ . To ease the training, we pad lines with fewer than  $k$  words and truncate lines that are longer than  $k$  words. By default,  $k = 30$ , which is sufficient to represent 98.4% of all lines in our dataset without truncating any words. Similarly, we fix the number of lines per trace to  $n$  by padding or truncating traces that are too short or long, respectively. By default,  $n = 100$ , which is shorter than most raw traces in our dataset, but covers most information relevant for localizing the buggy file. Each word is represented as a real-valued vector of length  $e = 100$ . To encode words into vectors, we pre-train Word2vec embeddings [16] on all traces. Intuitively, the embeddings assign a similar vector to words that occur in similar contexts.

The input to the line-level RNN is a matrix  $\mathbb{R}^{k \times e}$ . We use a bi-directional RNN that summarizes the sequence of words both in a forward and a backward pass and then concatenates both summaries into a single vector. The line-level RNN outputs a vector of length  $\gamma = 140$ , which summarizes the content of the line. Given

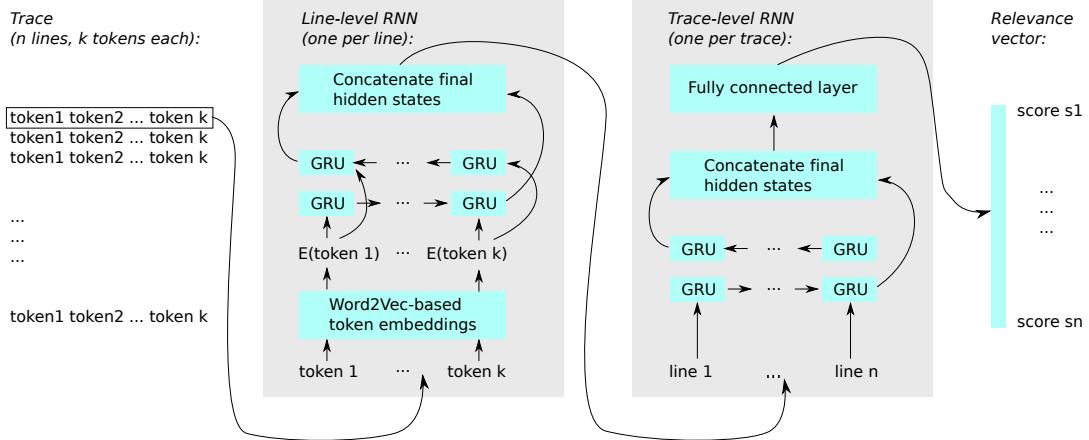


Figure 3: Neural trace-line model.

the continuous vector representation of each line, the trace-level RNN takes the sequence of lines encoded in a matrix  $\mathbb{R}^{l \times Y}$ . Similar to the line-level RNN, the trace-level RNN feeds the given sequence through a bi-directional RNN followed by a fully-connected layer. Finally, the model predicts the relevance vector  $r \in \mathbb{R}^n$ .

All parts of the neural trace-line model are trained jointly based on the ground truth of trace-relevance pairs. The rationale is to allow the network to find a representation of words and lines that is most suitable for addressing the line prediction problem.

#### 4 PREDICTING PATHS FROM TRACE LINES

The second sub-problem that Scafle addresses is to match the most relevant line of a crash trace, as predicted by the trace-line model, to file paths in the code base. This sub-problem comes with two main challenges. One challenge is to scale well to code bases that contain millions of files. Scafle partially addresses this challenge by focusing on a single line of a trace, instead of comparing each line with the code base. Another challenge is that the most relevant trace line may not mention the exact file path where the bug is located. Possible reasons are that a prefix of a file path may be missing because the crash trace uses a path relative to the crash location, or that the trace simply does not contain the file name.

Our approach to predict file paths from trace lines addresses the above challenges by formulating the problem as an information retrieval (IR) problem. While IR is usually concerned about retrieving a few out of many documents for a given query, we aim at retrieving a few out of many file paths for a given trace line. The second part of Scafle addresses the following problem:

*Definition 4.1 (Path prediction problem).* Given a trace line  $l$  and a set  $P$  of file paths in a code base, predict a ranked list  $(p_1, p_2, \dots)$  of file paths, with  $p_i \in P$ , such that  $l$  is most likely to refer to the highest ranked paths.

Inspired by IR techniques that retrieve documents for a given natural language query, Scafle represents both the trace line (i.e., query) and each file path in the code base (i.e., document) as a set of words. File paths are tokenized into words by path segments, and in a similar manner, the trace line is tokenized by punctuation after stripping line numbers. An IR-based search engine is then run on

the file paths to index the words appearing in them. Scafle allows this search engine to be generic – in our evaluation (Section 6), we used a search tool based on vector space embedding, similar to [20]. However, any off-the-shelf search tool could be plugged into the approach. For instance, we also experimented with the Okapi BM25 function, which is common in elastic search, and found the difference between the two to be negligible.

When indexing the corpus of file paths, most modern search engines down-weight words that are prevalent across the corpus, and up-weight words that are more distinctive. For the example in Figure 2(a), the query consist of the words “at dosstuff DoStuffBrowserController exitDoMoreStuff DoStuffBrowserController java”. Because “java” is a very common word, it gets down-weighted, while “DoStuffBrowserController” is more distinctive and hence up-weighted. The IR component used in Scafle weights words based on their tf-idf weight.

Once the search engine has indexed the corpus of file paths, Scafle queries it with words from the most relevant lines predicted by the trace-line model. In the end, the result of path prediction is a list of paths  $p_1, p_2, \dots$  returned from the search query, ranked by IR-based similarity of the paths to the query.

An alternative to our IR-based way of addressing the path prediction problem would be to simply return the best-matching file based on the number of overlapping tokens. However, this alternative approach would be misled by commonly occurring terms, such as underlying framework and library names. An IR-based search overcomes this problem (and a few others) by weighting terms up/down depending on their distinctiveness in the corpus.

#### 5 IMPLEMENTATION

We implement the trace-line model in PyTorch using gensim’s implementation of Word2vec. The Word2vec embedding layer contains 100 dimensions, the line-level RNN consists of two hidden layers with 70 GRU cells each, and the trace-level RNN has two layers of 250 GRU cells each. We use the sigmoid function for activation and mean-squared error as the loss function. The model is trained for 50 epochs using the Adam optimizer with a learning rate of 0.001. The path prediction part of Scafle is implemented using the standard tf-idf vectorizer in scikit-learn.

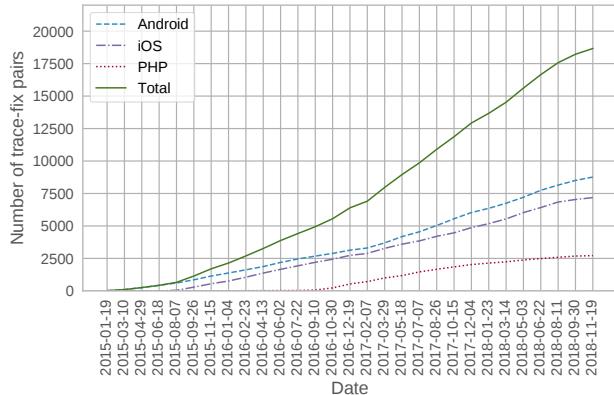


Figure 4: Number of crash traces in our dataset.

## 6 EVALUATION

We evaluate Scaffold with tens of thousands of crash traces and a code base consisting of millions of files in multiple programming languages. We address the following research questions:

- RQ 1: How effective is Scaffold at predicting bug locations?
- RQ 2: Given a raw trace, how effective is the learned trace-line model at identifying the most relevant lines?
- RQ 3: Why does Scaffold (sometimes not) work?
- RQ 4: How does Scaffold compare to existing approaches and to simpler baseline approaches?
- RQ 5: Is Scaffold efficient enough to scale to large code bases?

### 6.1 Experimental Setup

Our evaluation is based on tens of thousands of field crashes and their corresponding bug fixes. The data has been selected among crashes at Facebook over several years and comprises crashes from a diverse set of products. Field crashes observed in these products are automatically clustered to avoid inspecting the same problem multiple times. We use at most one representative of each such cluster, i.e., crashes in our dataset are unique. The crashes consist of Android crashes, such as the example in Figure 2(a), iOS crashes, and crashes in PHP code, such as the example in Figure 2(b), which contribute 46.8%, 38.7%, and 14.5% to the dataset, respectively. For each crash, we have a raw crash trace, which we feed into Scaffold without any preprocessing or parsing, except splitting the trace into lines. To establish ground truth data for bug localization, we associate crashes with bug fixing commits based on an issue tracker-like system at Facebook. Each bug fix changes one or more files, which we consider to be the bug location to predict. The minimum, mean, and maximum number of files changed in a bug-fixing commit is 1, 1.8, and 512, respectively.

Compared to similar setups in the literature [9, 12, 13, 18, 19, 21, 25, 31–33], there are two important differences: First, our dataset contains data from multiple projects and programming languages, which results in a more diverse set of crash traces. Second, our code base contains millions of files, i.e., it is at least two orders of magnitude larger than the largest previously considered code base.

To evaluate Scaffold in a realistic setup, we simulate using the approach at different points in time. Figure 4 shows the cumulative

number of crash traces used in the evaluation. At each point in time  $t$  shown on the horizontal axis, we simulate using Scaffold by training its model based on all data available at  $t$  and by predicting the bug locations for all crash traces that occur between  $t$  and  $t + 50$  days. For the prediction, we gather the set of all files in the code base at  $t + 50$  days and let the path prediction component of Scaffold predict which of these files need to be fixed. This setup is realistic, as it uses only past data to predict future bug locations. A possible, but less realistic alternative would be to randomly split all available data into a training and a validation set.

We evaluate Scaffold on two variants of the crash traces in our dataset: raw traces and stack traces. *Raw traces* contain the crash stack, additional telemetry, and information added by other tools that handle crashes at Facebook. The examples in Figure 2 are raw traces. Particularly, raw traces contain the output of a heuristic logic that is used to aggregate crash reports into groups. This output is then used by engineers to identify the files of the codebase relevant to the crash, and so it serves as a *de facto* bug localization method. The goal here is to evaluate whether in an industrial setting with additional information as in raw traces, Scaffold can still add value. We compare Scaffold with the heuristic logic alone in Section 6.5. *Stack traces* contain only the crash stack that we extracted from the raw traces, and are stripped of any other information. In Figure 2(a) and (b), the extracted stack traces begin at lines 2 and 31, respectively. The goal here is to evaluate Scaffold in a more “pure” setting where only crash stacks are available [25, 30, 33].

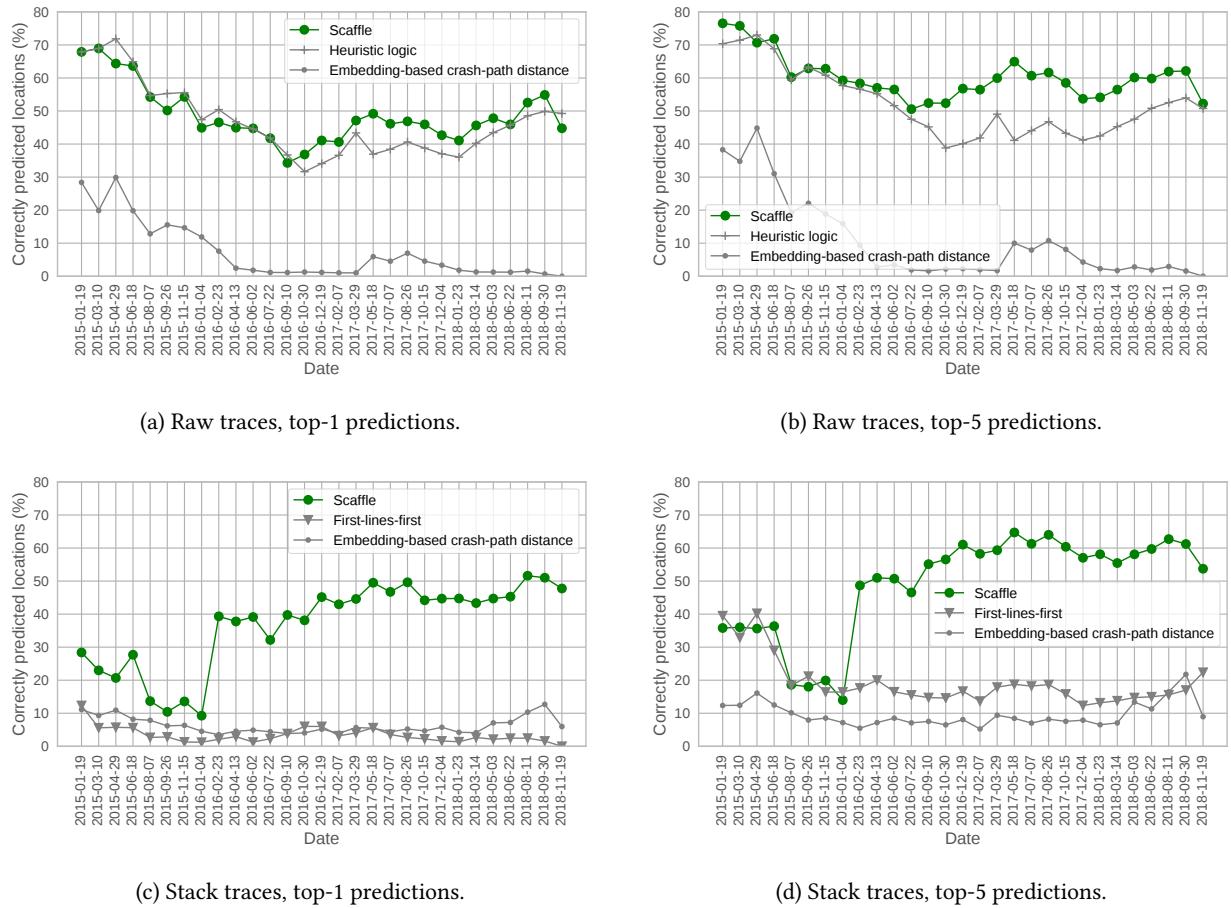
### 6.2 RQ 1: End-to-End Effectiveness

To measure how effective Scaffold is at predicting bug locations, we ask the approach to predict a ranked list of likely buggy files and then compare these files to those that have actually been changed by the developers. If the top predicted files include any of the actually changed files, we consider the prediction to be correct. The rationale is that pinpointing at least one of the files to modify is helpful in practice to identify which developer should handle a crash and which part of the code base the developer should focus on.

Figure 5 shows the percentage of correct predictions among the top-n predicted files. The plots on the left and right are for top-1 and top-5 predictions, respectively. The plots at the top and bottom are for raw traces and stack traces, respectively. The results vary over time because at each time step, a different model gets trained and because the crashes used for the evaluation vary from one 50-day period to another. The gray lines show baselines that we discuss in more detail in Section 6.5.

Overall, Scaffold is effective at selecting the bug location from millions of possible files. The model generally predicts between 40% and 60% of all bug locations as the top-1 prediction, and between 50% and 70% in the top-5 predictions. The results change over time for mostly two reasons. First, the composition of crashes in our dataset changes over time, as illustrated in Figure 4. The relative percentage of Android crashes decreases, while the percentage of crashes in iOS and PHP increases. In particular, it is only at some point in 2016 that the dataset starts to contain PHP crashes.<sup>1</sup> Second, for stack traces the model seems to require a certain number of training examples to unfold its full power. As evidenced by the steep

<sup>1</sup>The composition of crashes is not representative for software or crashes at Facebook, but merely a result of our data gathering process.



**Figure 5: Top-n accuracy of predicted bug locations. The green line is the Scaffle approach.**

increase in accuracy toward the beginning of 2016 in Figure 5(c) and (d), the model is much more effective once it has seen enough data.

### 6.3 RQ 2: Effectiveness of Trace-Line Model

To better understand Scaffle, we evaluate how effective the trace-line model is at predicting the most relevant lines in a crash trace. We use two metrics: Hit rate at n (hit@n) and mean reciprocal rank (MRR). Both metrics are computed w.r.t. the most relevant lines, as defined in the ground truth computed with Algorithm 1. The hit@n metric is the percentage of traces where the most-relevant line is among the top-n lines predicted by the model. For example, if the most relevant line is predicted as the second-most relevant line by the model, then this counts as a hit@3 but not as a hit@1. The MRR metric is computed by taking the predicted rank of the most relevant line, computing its reciprocal, and then averaging across all traces. For example, suppose there are only two traces and that the most relevant line of trace 1 and trace 2 is predicted at rank 1 and 4, respectively, then the MRR is  $\frac{1}{2} \cdot (\frac{1}{1} + \frac{1}{4}) = 0.625$ .

Figure 6 shows the results for the hit@n metric. Again, the green lines are for the Scaffle approach; all other lines are baselines discussed in Section 6.5. Solid lines show hit@1 results, dotted lines show the corresponding hit@5 results. The results roughly follow

the same patterns as the end-to-end results in Figure 5, confirming that the effectiveness of the trace-line model is key to the overall success of Scaffle. Notably, the model achieves a hit@5 rate above 80% for most points in time.

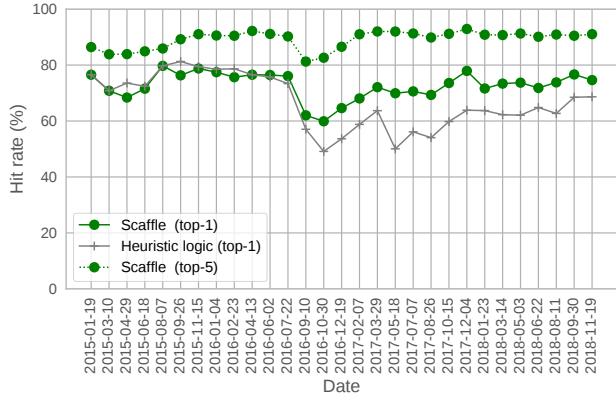
Figure 7 shows the results for the MRR metric. Again, we find the trace-line models to be highly effective, reaching MRR values of over 0.8 for raw traces and close to 0.8 for stack traces.

### 6.4 RQ 3: Why Does Scaffle Work?

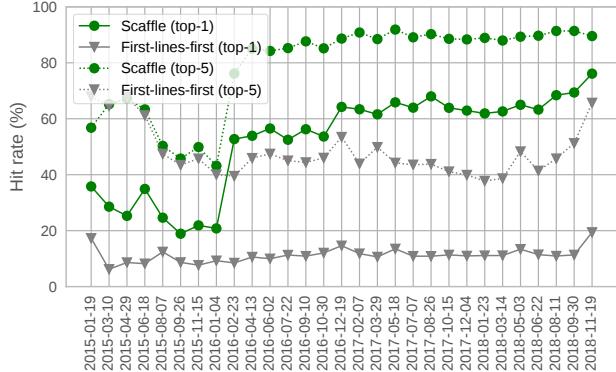
To better understand why the approach often is, and sometimes is not, able to identify the files relevant for fixing a bug, we manually inspect various traces and their corresponding bug locations. This inspection leads to the following observations.

*Files mentioned in traces.* Perhaps unsurprisingly, Scaffle is effective when the file that needs to be fixed is mentioned in the crash trace, e.g., because one of the functions in the file appears in a stack trace mentioned in the trace. In contrast, the approach cannot predict the correct bug location when the buggy file is not mentioned anywhere in the trace. The latter case may happen when the root cause and the manifestation of a bug are in different files.

*Partial information about files.* Even incomplete mentions of a file may be sufficient to enable Scaffle to localize it. For example, some crash traces mention the relevant file name, but not the complete



(a) Raw traces.

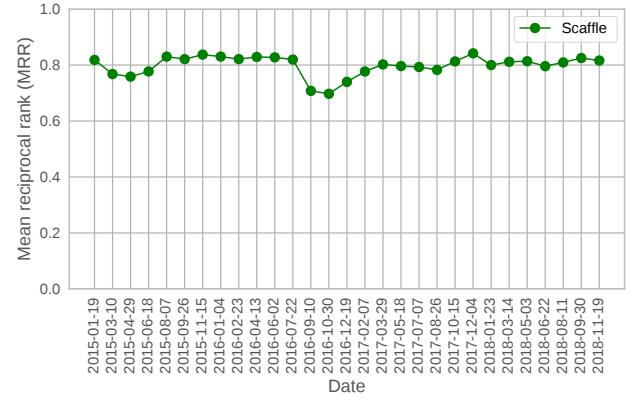


(b) Stack traces.

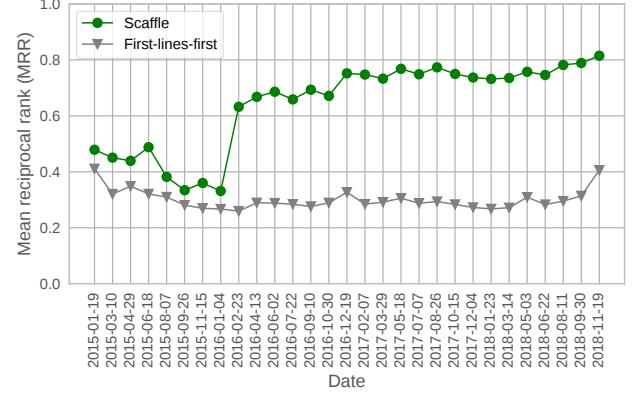
**Figure 6: Hit rate within the top-n predictions of most relevant lines. The green lines are for the Scaffold approach.**

path of the file. The reason may be that file paths on deployment devices differ from those in the code base or that the crash trace format does not include complete paths. Scaffold’s IR-based matching of lines and file paths exploits the fact that some high-entropy segments of a path, e.g., only the file name, often are sufficient to uniquely identify a path. In contrast, the approach sometimes fails to uniquely identify the correct file because multiple files with the same file name exist in different directories of the code base.

*Understanding the structure of traces.* The neural trace-line model has some “understanding” of the structure of raw crash traces. As illustrated by the examples in Figure 2, it identifies stack traces within the raw traces by giving lines that are part of stack trace generally higher relevance scores. Moreover, the model learns to identify the relevant lines within a stack trace by discarding stack frames unlikely to point to a bug location. For example, the model gives a relatively low score to those lines in Figure 2(a) that start with `at android.app`, because they refer to methods in the Android framework, i.e., code unlikely to cause a bug in the application. The model also learns to handle nested stack traces, such as the one in Figure 2(a), where one exception causes another. For such stack



(a) Raw traces.



(b) Stack traces.

**Figure 7: MRR of the trace-line model and a heuristic baseline. The green lines are for the Scaffold approach.**

traces, the model learns to search the most relevant lines in the inner-most exception, as this exception is the cause of the crash.

Instead of learning to understand raw traces and the stack traces contained in them, one could manually implement heuristic algorithms for parsing different trace formats. One of our baselines (Section 6.5) are a set of such heuristics. The main benefit of a learned model is that it can be obtained automatically and that it can be easily re-trained when trace formats evolve.

## 6.5 RQ 4: Comparison with Prior Work and Baselines

**6.5.1 End-to-End Classification.** Scaffold breaks down bug localization into two phases: predicting the most relevant lines in a crash trace and retrieving the most relevant files based on these lines. Prior work, e.g., by Kim et al. [9], proposes to directly predict the buggy file from a bug report or crash trace. The basic idea is to extract features of the crash trace, such as words appearing in the report, and train a classifier on past fixes to predict a set of relevant files to fix. To improve the precision of their model, they split the process into two phases: one that predicts if a crash report contains useful information, and one that actually predicts the file.

We implement the model described by Kim et al. [9] using a Naive Bayes and a Random Forest classifier. Unfortunately, the model suffers from acute scalability problems, and we were unable to train it on more than 10,000 files before running out of memory. The reason is the large number of file paths, each of which is a unique “class” for the classifier. Generally, classifiers scale well with the number of features, but not with the number of classes. Since an end-to-end classification approach, such as [9], considers each file as a class, it does not scale to code bases with millions of files.

**6.5.2 End-to-End Information Retrieval.** IR-based bug localization is one of the most prevalent approaches proposed in prior work [13, 17, 19, 21, 27, 33]. We compare Scafle with a scalable variant of these approaches that avoids running a source-level analysis of all files in the code base. This end-to-end, IR-based approach considers the file paths as documents, similar to Scafle, and matches them against the given crash trace. To implement this baseline, we use the path prediction model in Section 4 to embed file paths into a tf-idf based vector space. Given a crash trace, the baseline embeds the entire trace into the same vector space, and reports those file paths as most relevant for the crash that have the lowest cosine distance to the embedded crash trace.

The result of this baseline is plotted in Figure 5 (‘Embedding-based crash-path distance’). The baseline performs poorly compared to Scafle, especially as the number of crashes and file paths grow over time. The reason is that the efficiency of IR techniques relies largely on the quality of the query provided for search. However, if the query is the entire crash trace, it contains significant amounts of noise, which adversely affects the accuracy of retrieval. The comparison with this baseline shows that decomposing the bug localization problem by first predicting the most relevant line of a crash trace is key to achieving good overall accuracy.

**6.5.3 First Lines First.** A simple baseline to identify the most relevant lines in a given stack trace is to assume that the lines in the stack trace are sorted by descending relevance. This baseline, which we call *first-lines-first*, matches an assumption made in prior work on bug localization [19]. All figures for stack traces show the results for this baseline as a gray line, i.e., Figure 5(c) and (d), Figure 6(b), and Figure 7(b). We find the first-lines-first baseline to perform clearly worse than Scafle because the root cause of a crash is not always mentioned in one of the first lines.

**6.5.4 Heuristic Logic.** As an industrially deployed baseline, we consider a heuristic logic that aggregates crash reports at Facebook. It is based on a series of manually designed and tuned rules that extract relevant strings from the crash, such as the exception message, type, or parts of stack frames. The output is then added to the raw trace and is used to group together crashes that have the same output from the logic. Engineers use it to gain hints about files that are relevant to the crash. To measure whether Scafle adds any value on top of the heuristic logic, we consider the heuristic logic as a stand-alone baseline and show its effectiveness in all figures for raw traces as a gray line, i.e., in Figure 5(a) and (b), and in Figure 6(a). For the end-to-end prediction, we find this baseline to be less effectiveness than Scafle. In other words, the model learns to override the suggestions of the heuristic logic by predicting other lines in a raw trace than the result of the logic as the most relevant. This shows

that even in an industrial setting, Scafle can be used to augment any existing localization methods. Another advantage of Scafle is to predict a ranked list of likely relevant lines and likely buggy files. For example, as shown in Figure 6(a), the top-5 predictions have a higher hit rate than the heuristic baseline.

## 6.6 RQ 5: Efficiency

To evaluate the efficiency of Scafle, we distinguish between one-time efforts and per-trace efforts. The computationally most expensive one-time effort is to train the trace-line model. Depending on the amount of training data, the training takes up to three hours on a single machine equipped with a standard GPU. The time to predict the bug location for a given crash trace is the sum of the time needed by the two main steps of Scafle. Querying the model with a given trace generally takes less than a second. The IR-based matching of the top-most predicted lines against all file paths in the code base takes up to several seconds per line. This amount of time is acceptable in practice because Scafle can run in the background and report its suggestions to developers once it is done. Overall, we conclude that Scafle is efficient and scalable enough to run in an industrial deployment, even for large-scale code bases.

## 7 DISCUSSION

This section discusses some limitations and alternative designs of the approach. One limitation is that Scafle predicts only one file out of possibly many files that cause a crash. On average over all crashes used in the evaluation, we find that 1.8 files are modified to fix a crash. That is, finding one of the relevant files covers a large fraction of all relevant files in practice. Even if multiple files need to be modified, predicting one of them still provides a good starting point to find the right team or developer to handle the crash, and to find the remaining files using other techniques.

The prediction accuracy of Scafle ranges between 40% and 70%, which raises the question whether it is high enough to be useful in practice. While a higher accuracy would certainly be desirable, the current result can contribute some value. One piece of evidence is that Scafle outperforms the heuristic logic that is currently used at Facebook to help developers handle crashes (Section 6.5.4). Given that crash-based bug localization is a hard yet practically relevant problem, we envision future work to further improve the current result. One promising direction may be to combine a scalable technique, such as Scafle, with a more expensive technique that reasons only about selected subsets of the code base.

Our evaluation learns a single trace-line model across all crash traces, irrespective of the specific product that has crashed or the programming language used to write the crashing code. Instead, one could train a separate model for different subsets of our dataset. There are at least two reasons why we choose a single-model approach. One reason is that the crash traces contain language-independent information added by tools that pre-process traces at Facebook. By learning a single model across all crashes, the model can generalize better and learn patterns that hold across products and languages. Another reason is that the set of products, programming languages, etc. is constantly evolving. A single model eliminates the need to build multiple pipelines and is more robust toward an evolving code base.

Scaffold is designed with ultra-large scale, heterogeneous code bases in mind. Smaller software projects may not have any or not enough information about past crashes available. For such projects, our approach may not work well, because training an effective trace-line model requires a sufficiently large training dataset. For such smaller projects, the code base may be small enough to analyze each source file individually, making other existing techniques (Table 1) a viable alternative.

## 8 RELATED WORK

*Localizing Bug Locations.* The most closely related line of work also predicts bug locations based on some evidence of a bug, such as a bug report or a stack trace, as summarized in Table 1. Our work, shown in the last row, differs by taking raw crash traces as the input, without making any assumptions about their format. Almost all other techniques also differ from ours by focusing on a single programming language and by requiring some form of parsing and code analysis of all files in the code repository. The work by Wang et al. [25] is the exception, as it also targets multiple languages and does not require parsing or analyzing the code. It differs from Scaffold in multiple ways: First, their approach learns a model that, given features of a specific file, predicts the probability that the file contains the bug. This design implies that predicting the buggy file requires querying the model with all files in the code base, which does not scale well to code bases with millions of files, such as the one that motivates our work. Second, their approach assumes that file names mentioned in a crash trace can always be resolved to file paths in the code base. We find this assumption to sometimes be violated, e.g., because the path of a file on a client device differs from the path in the code base. Third, their approach expects a set of stack traces as the input, whereas Scaffold works well given only a single crash trace. Some of the work listed in Table 1 uses inputs beyond bug reports or crash traces, e.g., by also considering meta-data from version histories, such as files involved in fixing past bugs [13, 32]. Scaffold does not require this kind of meta-data, but could possibly benefit from it.

*Localizing Bug-Inducing Changes.* A related problem is to localize which commit is causing a bug. Locus addresses this problem in an IR-based approach that compares the words associated with a commit and the words in a bug report [26]. Orca also takes an IR-based approach and reports deploying a system for a large-scale, distributed, industrial system [2]. Their focus is on handling the frequent re-builds of the system through a build provenance graph. ChangeLocator takes a learning-based approach that ranks all commits that change at least one method that appears in a stack trace [29]. Similar to most techniques in Table 1, ChangeLocator relies on statically analyzing the code base, which Scaffold avoids.

*Clustering and Prioritizing Crashes.* The potentially large number of crashes revealed by fuzz-testing or widely deploying software has motivated work on clustering crashes, e.g., based on similarities of stack traces [4, 5], on repairs that prevent a crash [23], or other heuristics [11]. Once crashes are clustered, Castelluccio et al. [3] propose to help understanding the crash by identifying features that are unique to a cluster. Another line of work prioritizes crashes, e.g., based on the distribution of occurrences among users [8] or

based on a prediction of how likely a crash will occur for other users [10]. All this work is orthogonal to the problem addressed here, and could be used before localizing the bug location for a crash with Scaffold.

*Other Related Work.* Many techniques for automated program repair [14] rely on localizing where to fix a bug. Scaffold could serve as a starting point for repair of bugs that manifest through crashes. The broader problem of bug localization has received significant attention. We here restrict our discussion to crash-based localization and refer to a survey [28] for detailed discussion. The main difference to coverage-based bug localization [1, 6, 15] is that Scaffold does not require any coverage information, but only a crash trace that manifests the bug. Wang et al. [24] study whether IR-based bug localization for given bug reports simplifies debugging. Our setup differs by considering machine-generated, raw crash traces instead of (at least partially) human-written bug reports. Our trace-line model exploits one of their observations, that “some stack traces and test cases contain many class names and method names, and only a small subset of the names are closely related to the bug” [24]. Jonsson et al. [7] address the problem of assigning a bug report to a developer team. They also focus on large-scale deployment, but with dozens of teams instead of millions of files to choose from.

## 9 CONCLUSIONS

This paper presents Scaffold, the first technique for automated, crash-based bug localization in ultra-large scale, heterogeneous code bases. The key idea is to decompose the problem into two simpler sub-problems: (1) Identifying the most relevant lines in a raw crash trace, which we address through a neural trace-line model, and (2) Matching these lines with file paths in the code base, which we address through a scalable, IR-based search. The approach is language-independent, as it does not assume a specific trace format, but instead learns from crashes fixed in the past. Our evaluation applies Scaffold to a code base that is at least two orders of magnitude larger than any previous work. We find the approach to be effective at identifying files to fix, despite having to choose among millions of code files written in several programming languages. Notably, Scaffold outperforms several non-trivial baselines, including end-to-end classification, an end-to-end IR-based search, and industrially used heuristics. Overall, Scaffold provides a practical technique for pinpointing the files to consider in a large-scale code base, which helps assigning crashes to the appropriate team or developer, and may serve as a starting point for automated program repair.

## ACKNOWLEDGMENTS

Parts of this work were supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No. 851895) and by the German Research Foundation within the ConcSys and Perf4JS projects.

## REFERENCES

- [1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
- [2] Ranjita Bhagwan, Rahul Kumar, Chandra Shekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in Large-Scale Services. In

- [3] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. 2017. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 717–726. <https://doi.org/10.1145/3106237.3106306>
- [4] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 1084–1093. <https://doi.org/10.1109/ICSE.2012.6227111>
- [5] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. 333–342. <https://doi.org/10.1109/ICSM.2011.6080800>
- [6] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. 467–477. <https://doi.org/10.1145/581339.581397>
- [7] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578. <https://doi.org/10.1007/s10664-015-9401-9>
- [8] Foutse Khomh, Brian Chan, Ying Zou, and Ahmed E. Hassan. 2011. An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*. 261–270. <https://doi.org/10.1109/WCRE.2011.39>
- [9] Dongsun Kim, Yida Tao, Sungjun Kim, and Andreas Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Trans. Software Eng.* 39, 11 (2013), 1597–1610. <https://doi.org/10.1109/TSE.2013.24>
- [10] Dongsun Kim, Ximming Wang, Sungjun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. 2011. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *IEEE Trans. Software Eng.* 37, 3 (2011), 430–447. <https://doi.org/10.1109/TSE.2011.20>
- [11] Kinshuman Kinshumann, Kirk Glerner, Steve Greenberg, Gabriel Aul, Vince R. Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. 2011. Debugging in the (very) large: ten years of implementation and experience. *Commun. ACM* 54, 7 (2011), 111–116. <https://doi.org/10.1145/1965724.1965749>
- [12] Anil Koyuncu, Tegawende F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. *IEEE Transactions on Software Engineering* (2019).
- [13] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 476–481. <https://doi.org/10.1109/ASE.2015.73>
- [14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. <https://doi.org/10.1145/3318162>
- [15] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 82–92. <https://doi.org/10.1145/3180155.3180242>
- [16] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionalities. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.
- [17] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. 151–160. <https://doi.org/10.1109/ICSME.2014.37>
- [18] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Lawrence, KS, USA, November 6-10, 2011. 263–272. <https://doi.org/10.1109/ASE.2011.6100062>
- [19] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 621–632. <https://doi.org/10.1145/3236024.3236065>
- [20] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 31–41.
- [21] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 345–355. <https://doi.org/10.1109/ASE.2013.6693093>
- [22] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2017. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 209–220. <https://doi.org/10.1109/ICSE.2017.27>
- [23] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 612–622. <https://doi.org/10.1145/3238147.3238200>
- [24] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 1–11. <https://doi.org/10.1145/2771783.2771797>
- [25] Shaohua Wang, Foutse Khomh, and Ying Zou. 2013. Improving bug localization using correlations in crash reports. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. 247–256. <https://doi.org/10.1109/MSR.2013.6624036>
- [26] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 262–273. <https://doi.org/10.1145/2970276.2970359>
- [27] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 181–190.
- [28] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. <https://doi.org/10.1109/TSE.2016.2521368>
- [29] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. Change-Locator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23, 5 (2018), 2866–2900. <https://doi.org/10.1007/s10664-017-9567-4>
- [30] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sungjun Kim. 2014. CrashLocator: locating crashing faults based on crash stacks. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 204–214. <https://doi.org/10.1145/2610384.2610386>
- [31] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2015. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. *IEEE Transactions on Software Engineering* 42, 4 (2015), 379–402.
- [32] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 689–699. <https://doi.org/10.1145/2635868.2635874>
- [33] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 14–24. <https://doi.org/10.1109/ICSE.2012.6227210>



# Abstracting Failure-Inducing Inputs

Rahul Gopinath

rahul.gopinath@cispa.saarland  
CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany

Alexander Kampmann

alexander.kampmann@cispa.saarland  
CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany

Nikolas Havrikov

nikolas.havrikov@cispa.saarland  
CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany

Ezekiel O. Soremekun  
ezekiel.soremekun@cispa.saarland  
CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany

Andreas Zeller  
zeller@cispa.saarland  
CISPA Helmholtz Center for  
Information Security  
Saarbrücken, Germany

## ABSTRACT

A program fails. Under which circumstances does the failure occur? Starting with a single failure-inducing input (“The input ((4)) fails”) and an input grammar, the DDSET algorithm uses systematic tests to automatically generalize the input to *an abstract failure-inducing input* that contains both (concrete) terminal symbols and (abstract) nonterminal symbols from the grammar—for instance, “((*expr*))”, which represents any expression *(expr)* in double parentheses. Such an abstract failure-inducing input can be used (1) as a *debugging diagnostic*, characterizing the circumstances under which a failure occurs (“The error occurs whenever an expression is enclosed in double parentheses”); (2) as a *producer* of additional failure-inducing tests to help design and validate fixes and repair candidates (“The inputs ((1)), ((3 \* 4)), and many more also fail”). In its evaluation on real-world bugs in JavaScript, Clojure, Lua, and UNIX command line utilities, DDSET’s abstract failure-inducing inputs provided to-the-point diagnostics, and precise producers for further failure inducing inputs.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging;
- Theory of computation → Grammars and context-free languages; Active learning.

## KEYWORDS

debugging, failure-inducing inputs, error diagnosis, grammars

### ACM Reference Format:

Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. 2020. Abstracting Failure-Inducing Inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397349>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA ’20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397349>

## 1 INTRODUCTION

Having to deal with software failures is the daily bread of software developers—frequently during development and testing, (hopefully) less so during production. Since failures are caused by concrete inputs, but must be fixed in abstract code, developers must determine the *set of inputs that causes the failure*, such that the fix applies to precisely this set. This is important, as an incorrect characterization leads to incomplete fixes and unfixed bugs.

As an example, consider a calculator program that fails given the input in Fig. 1a. To identify the *set of failure-inducing inputs*, the developer must ask herself: Is the error related to parenthesized expressions? Any parenthesized expression? Doubled parentheses? Or just nested parentheses? Or is the error related to addition, multiplication, or the combination of both? Each of these conditions induces a different set of inputs, and to fix the bug, the developer has to identify the failure-inducing set as precisely as possible—typically following the scientific method through a series of experiments, refining and refuting hypotheses until the failure-inducing set is precisely defined.

1 + ((2 \* 3 / 4))

(a) Failure-inducing input (b) Abstract failure-inducing input

Figure 1: Input for the calculator program.

```

⟨start⟩ ::= ⟨expr⟩
⟨expr⟩ ::= ⟨int⟩ | ⟨var⟩
        | ⟨prefix⟩ ⟨expr⟩ | "(" ⟨expr⟩ ")" | ⟨expr⟩ ⟨op⟩ ⟨expr⟩
⟨op⟩ ::= "+" | "-" | "*" | "/" |
⟨prefix⟩ ::= "+" | "-"
⟨int⟩ ::= ⟨digit⟩⟨int⟩ | ⟨digit⟩
⟨var⟩ ::= ⟨chars⟩⟨char⟩ | ⟨chars⟩
⟨digit⟩ ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
⟨char⟩ ::= "a" | "b" | "c" | "d" | "e" | "f"

```

Figure 2: Simple expression grammar

In this paper, we present an approach that fully *automates* the process of characterizing failure-inducing inputs. The DDSET algorithm<sup>1</sup> starts with a single failure-inducing input (such as the one above), a precise characterization of the failure and an input grammar (in our case, representing arithmetic expressions, as in Fig. 2). Using these three, it runs a number of experiments (tests) to derive an abstract failure-inducing input that abstracts over the original input, replacing concrete terminal symbols (characters) by abstract nonterminals from the grammar. In our example, such an abstract failure-inducing input produced by DDSET would be the one in Fig. 1b, where  $\langle \text{expr} \rangle$  is actually *any* expression as given by its grammar expansion rule. Formally, an abstract failure-inducing input represents the set of all inputs obtained by expanding the nonterminals it contains—for Fig. 1b, *any* expression contained in double parentheses.

Can we determine that *all* expansions are failure-inducing? This would require a fair amount of symbolic analysis, and be undecidable in general. Instead, DDSET exploits the fact that the abstract failure-inducing input can be used as a *producer* of inputs. DDSET thus instantiates it to numerous concrete test inputs; the abstract failure-inducing input is deemed a valid abstraction only if *all* its instantiations reproduce the original failure.

The concept of an abstract failure-inducing input and how to determine it are the original contributions of this paper. Such an abstract failure-inducing input can be used for:

**Crisp failure diagnostics.** Abstract failure-inducing inputs such as  $((\langle \text{expr} \rangle))$  precisely characterize the condition under which the failure occurs: “The failure occurs whenever an expression is surrounded by double parentheses”. Their simplicity is by construction, as they are at most as long (in the number of tokens) as the shortest *possible input that reproduces the failure*. Should a developer prefer a number of examples rather than the abstraction, an abstract failure-inducing input can be used to produce *minimal inputs* similar to delta debugging [19]; in our case, this would be inputs such as  $((0))$ ,  $((1))$ , etc.

**Producing additional failure-inducing inputs.** In manual debugging as well as automated repair, a common challenge is to validate a fix: How do we know we fixed the cause and not the symptom? An abstract failure-inducing input as produced by DDSET can serve as *producer* to generate several inputs that all trigger the failure. In our case, these would include inputs such as  $((1))$ ,  $((-3))$ ,  $((5 + 6))$ ,  $((8 * (3 / 4)))$ ,  $((((9) * (10))))$ , or  $((((11 / 12) + -13)))$ . All these inputs trigger the same original failure, and any fix should address them all. A badly designed fix that would cover only a subset (say, some single symptom of the original input such as “no digits with two levels of parenthesis”) would be invalidated in an instant.

How does DDSET produce an abstract failure-inducing input? DDSET uses the given input grammar (e.g. Fig. 2) to *parse* the input into a *derivation tree*, representing the input structure by means of syntactic categories; Fig. 3 shows the derivation tree for our example. DDSET then applies the following steps:

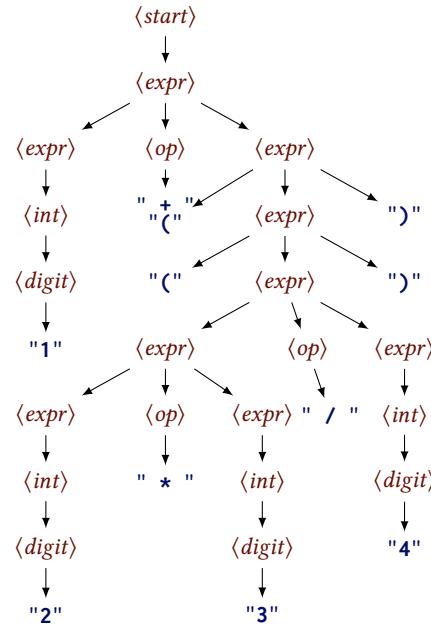


Figure 3: A derivation tree for  $1 + ((2 * 3 / 4))$

(1) **Reduction.** DDSET first *reduces* the input to a minimal failure-inducing input. As the input grammar is available, we can make use of efficient syntax-based reduction [18] rather than lexical delta debugging [19]. For effective reduction, these algorithms require a precise test condition that either

- (a) identifies that the input produced was semantically invalid
- (b) identifies that the input while semantically valid, but failed to reproduce the error, or
- (c) identifies that the input succeeded in reproducing the failure.

In our case, the input  $((3))$  is the result of reducing  $1 + ((2 * 3 / 4))$ .

(2) **Abstraction.** Even given a simplified input such as  $((4))$ , we do not know which elements cause the failure—is it the first parenthesis, both of them, or the number 4? To determine this, DDSET makes use of the same test procedure used by the *reduction* step, DDSET uses this test procedure to determine which concrete symbols can be replaced by other input fragments while still producing failures. In our case, it turns out that in the derivation tree of  $((4))$  (Fig. 5), the symbol 4 can actually be replaced by any  $\langle \text{digit} \rangle$ ,  $\langle \text{int} \rangle$ , and even  $\langle \text{expr} \rangle$ ; any expansion still produces failures. As a result, we obtain the derivation tree in Fig. 6, which expands into the *concrete failure-inducing input*. Hence, we characterize the abstract failure-inducing input as  $((\langle \text{expr} \rangle))$ .

(3) **Isolating Independent Causes.** Say the input was  $((1)) + ((2 * 3)) - ((5))$  and the failure cause was at least a pair of doubled parenthesis. Then, a naive abstraction will find the following fragments can be replaced by arbitrary expressions  $\langle \text{expr} \rangle + ((2 * 3)) - ((5))$ ,  $((1)) + \langle \text{expr} \rangle - ((5))$ , and  $((1)) + ((2 * 3)) - \langle \text{expr} \rangle$  while still

<sup>1</sup>DDSET = Delta debugging for input sets

inducing failures. This will lead to the erroneous conclusion that the final abstraction is  $\langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle$ . Our *isolation* step will correctly extract  $((\langle \text{expr} \rangle)) \langle \text{op} \rangle ((\langle \text{expr} \rangle))$ . Steps 2 and 3 work recursively until all independent causes are identified.

- (4) **Sharing Abstraction.** Given an input as  $a + a$ , with the failure condition being the repetition of a variable, the previous *abstraction* and *isolation* steps will find that the first variable  $a$  cannot be replaced with another arbitrary variable. The *sharing abstraction* step identifies such parts and verifies that all such parts can together be replaced by a shared abstraction. The resulting abstract failure-inducing input  $\langle \$var1 \rangle \langle \text{op} \rangle \langle \$var1 \rangle$  precisely identifies  $\langle \$var1 \rangle$  as a string that is shared.
- (5) **Handling Lexical Tokens.** A number of programming languages use *lexers* that skip over whitespace and comments. Further, program syntax may include optional elements such as annotations that are present in the derivation tree<sup>2</sup> and hence, the abstract pattern derived resolves to an empty string in the minimized input string. Finally, due to the way parsers and lexers work, tokens such as *begin* are represented by a nonterminal element such as  $\langle \text{BEGIN} \rangle$ , which can have only a single possible value. As these do not help developers, we identify these elements and replace them with their value in the compact representation.

The result of these steps is a **derivation tree** where *abstract*, *shared* and *invisible* nodes are marked, and a **compact representation** of this derivation tree as an abstract failure-inducing input. The compact representation is useful for developers while the derivation tree can be used as a producer for inputs.

In all that, the abstract failure-inducing inputs produced by DDSET capture dependencies even for complex input languages. Fig. 4a shows a failure-inducing input for the *Rhino* JavaScript interpreter. Fig. 4b shows its abstract representation as produced by DDSET. We see that instead of *baz*, we can have *any* identifier (as long as it is shared) and *any* variable declaration. The instantiations of this abstraction can produce numerous test cases that all help ensuring a proper fix.

In the remainder of this paper, we follow the steps of our approach, detailing them with the calculator example. After introducing central definitions (Section 2), we detail the steps of DDSET, namely reduction (Section 3), abstraction (Section 4), isolation (Section 5) identifying shared parts (Section 6), and identifying invisible elements (Section 7). We discuss properties and limitations of DDSET (Section 9). In our evaluation (Section 10), we apply DDSET on a range of bugs and subjects, assessing its effectiveness. After discussing threats to validity (Section 11) and related work (Section 12), Section 13 closes with conclusion and future work.

## 2 DEFINITIONS

In this paper, we use the following terms:

**Input.** A contiguous sequence of symbols fed to a given program. That is for our example,  $1 + ((2 * 3 / 4))$ , the symbols  $"1" " + " "(" "(" "2" " * " "3" " / " "4" ")" "$  form our input.

<sup>2</sup>These are marked as *SKIP* in ANTLR grammars.

**Alphabet.** The *alphabet* of the input accepted by a given program is the set of all non divisible symbols that the program can accept. In our example, the digits ("0" to "9"), operators ("+" "-" "\*" "/" ), prefixes ("+" "-") and parenthesis ("(" ")") forms the alphabet.

**Terminal.** An input symbol from the alphabet. These form the leaves of the derivation tree. For example, "2" is a terminal and so is "+".

**Nonterminal.** A symbol outside the alphabet that has a grammar definition. These form the internal nodes of the derivation tree. From our example,  $\langle \text{expr} \rangle$  is one of the nonterminals.

**Context-Free Grammar.** A set of recursive rules that is used to describe the structure of input. The context-free grammar is composed of a set of nonterminals and corresponding definitions that define the structure of the nonterminal. Each definition consists of multiple rules that describe alternative ways of defining the nonterminal. Fig. 2 describes a context-free grammar for an expression language.

We assume that the context-free grammar is given, and that it can parse the input to a<sup>3</sup> *derivation tree*<sup>4</sup>.

**Rule.** A finite sequence of terminals and nonterminals that describe an expansion of a given nonterminal. For our example,  $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle$  is one of the rules that defines the nonterminal  $\langle \text{expr} \rangle$  in the grammar.

**Derivation Tree.** A *derivation tree* is an ordered tree that describes how an input string is parsed by the rules in the grammar. Fig. 3 shows a derivation tree built by parsing  $1 + ((2 * 3 / 4))$  using the context-free grammar in Fig. 2.

**Predicate.** A test *predicate* determines if the given input is able to reach and reproduce the failure condition. If the input was not legal, that is, the input is invalidated by checks on the input before the predicate is reached, then the result is semantically UNRESOLVED. If the input reaches the predicate, and the failure condition is reproduced, the result is FAIL. If not, the result is PASS. Note that at all times, inputs are syntactically valid—that is, each input conforms to the context-free grammar. These follow the original definitions [19].

**Compatible Node.** A node is *compatible* to another if both have the same nonterminal. E.g. the root node for the expression  $2 * 3$  is an  $\langle \text{expr} \rangle$ . Similarly, the root node for the expression  $2 * 3 / 4$  is also an  $\langle \text{expr} \rangle$ . Hence, these two nodes are compatible.

**Compatible Tree.** A tree is *compatible* to another if both have compatible root nodes.

**Generated Compatible Tree.** One may randomly *generate compatible trees* given a nonterminal by the following production process:

- (1) Stochastically choose one of the rules from the definition corresponding to the nonterminal. The terminals and nonterminals in the rule form the immediate children of the root node of the generated tree.

<sup>3</sup>We assume that there is one canonical derivation tree. That is, no ambiguity, or in the case of ambiguity, and all derivation trees are semantically valid, we simply choose one tree. If not all derivation trees are semantically valid, we assume that there exists a procedure to identify the correct tree.

<sup>4</sup>If not, there are two choices: fix the input or the grammar. A technique like lexical *ddmax* would isolate the failure-inducing input in context and give sufficient hints to debug the input or the grammar such that the input can be parsed.

```

1 var {baz: baz => {}} = baz => {};
      var {<$Id1>:<$Id1> => {} } = <variableDeclaration>;
2   (a) Failure-inducing input
      (b) Abstract failure-inducing input for Fig. 4a

```

Figure 4: Issue 385 of the Rhino JavaScript interpreter.

- (2) For each nonterminal in the rule, stochastically choose a rule from the corresponding definition in the context-free grammar.  
(3) Continue the process until no nonterminals are left.

**Generated String.** Given a randomly generated compatible tree, the corresponding string representation is called the *generated string*. There can be an infinite number of generated strings corresponding to a nonterminal if the nonterminal is recursive (i.e. the nonterminal is reachable from itself).

**Reachable Nonterminal.** A nonterminal  $a$  is *reachable* from another nonterminal  $b$  if  $a$  is reachable from any of the rules in the definition of  $b$ . A nonterminal is reachable from a rule if (1) that nonterminal is present in the rule or  
(2) that nonterminal is reachable from any of the nonterminals in the rule.

From our expression grammar, the nonterminal  $\langle op \rangle$  is reachable from the nonterminals  $\langle expr \rangle$  and  $\langle start \rangle$ . The nonterminal  $\langle expr \rangle$  is reachable from itself and  $\langle start \rangle$ .

**Subtree.** For any given node in a tree, a *subtree* refers to the tree rooted in any of the reachable nonterminals from that node.

**String Representation.** For any given subtree, the corresponding *string representation* is the string fragment from the original input that corresponds to the subtree.

**Compact Representation.** For any given tree, the corresponding *compact representation* is the abstract string representation where the string representation of nodes marked as abstract are the corresponding nonterminal, shared nodes are represented by a parametrized nonterminal, and invisible nodes are represented by their corresponding string.

**1-minimal Input.** An input string is *1-minimal* if the predicate indicates that the string causes the failure, and removing any one symbol from the input no longer causes the failure.

**1-minimal-tree Input.** An input has a *1-minimal-tree* as its derivation tree if there is no node in the derivation tree that can be further simplified by the given reducer. Note that the definition of *1-minimal-tree* is dependent on the reduction algorithm used.

### 3 REDUCTION

Let us now get into the details of DDSET. As discussed in Section 1, DDSET starts with *reducing* the given input to a minimal input. As a reminder, as input to DDSET, we have

- the *predicate*—in our case, a program that fails whenever there are doubled parentheses in the input. For the sake of simplicity, we assume that the “calculator” program simply matches the input against a regular expression

$/ .*\[(\][\(\].*\[\)]\].*/$

—that is, it will fail on any input that contains a pair of opening and closing double parentheses, and pass otherwise.

- the *failure-inducing input*—in our case, the input  
 $1 + ((2 * 3 / 4))$
- the *input grammar*, as shown in Fig. 2.

DDSET uses the *Persees* reducer from Sun et al. [18]. We provide a brief overview of the Persees reducer. We first parse the input using the grammar provided, which results in a derivation tree (Fig. 3). The reduction algorithm (Algorithm 1) accepts this derivation tree and minimizes it to a minimal tree (Fig. 5).

---

#### Algorithm 1 The Persees reduction algorithm

---

```

function REDUCTION(string, grammar, predicate)
  dtree  $\Leftarrow$  parse(string, grammar)
  p_q  $\Leftarrow$  priority_queue((dtree,  $\emptyset$ ))
  while p_q  $\neq \emptyset$  do
    dtree, path  $\Leftarrow$  p_q.pop()
    snode  $\Leftarrow$  dtree.get(path)
    trees  $\Leftarrow$   $\emptyset$ 
    compatible_nodes  $\Leftarrow$  snode.get_all_nodes(snode.key)
    if  $\emptyset \in$  grammar[snode.key] then
      compatible_nodes.append( $\emptyset$ )
    end if
    for node  $\in$  compatible_nodes do
      ctree  $\Leftarrow$  dtree.replace(path, node)
      if predicate(ctree.to_s) == FAIL then
        trees.append((ctree, path))
      end if
    end for
    if trees  $\neq \emptyset$  then
      tree  $\Leftarrow$  minimal(trees)
      p_q.insert(tree, path)
    else
      for child  $\in$  nonterminals(snode.children) do
        p_q.insert(dtree, child.path)
      end for
    end if
  end while
  return dtree
end function

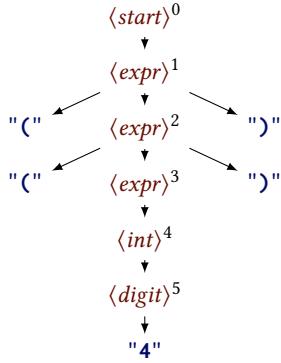
```

---

We start by maintaining a priority queue of tuples. The first item in the tuple is a derivation tree and the second item in the tuple is a path to a particular node in that derivation tree. We next add the full derivation tree and a path to the root node ( $t_0, p_0$ ) as the first item in the priority queue. The tree priority is determined by the number of terminal symbols (leaf nodes) on the derivation tree, followed by number of terminal symbols on the subtree that is indicated by the path. That is the shortest token sequence is at the top of the priority queue.

Next, DDSET performs the following steps in a loop.

- Extract the top tuple. It contains a complete derivation tree and the path to a subtree from that tree.



**Figure 5: The derivation tree for  $((4))$ . Nodes are annotated with numbers for easier reference.**

- (2) Given the subtree, identify the *compatible reachable nodes* from the root node of the subtree, ordered by their depth (shallowest first). These are *alternative trees* that we can replace the current subtree with a high chance of reproducing the failure. If the grammar allows the current node to be empty, then an empty node is added to the compatible nodes with a depth 0.
- (3) For each tree in the alternative, replace the current tree with the tree in the alternative producing a new derivation tree. Collapse the new derivation tree to its corresponding string representation, and check if the predicate confirms reproducing the failure. Collect every such alternative tree, and identify the tree that produces the smallest input (`minimal()`). Generate tuples for this tree – the first element is the derivation tree corresponding to the new input string, and the path is the same as the current path. Add this tuple to the priority queue. If we could find a smaller input in this step, go back to the first step.
- (4) If no smaller inputs could be found, generate new tuples by using the same derivation tree, but with paths that correspond to the children of the current node. Add the new tuples to the priority queue. Then go back to the first step.
- (5) The loop ends when the priority queue is empty.

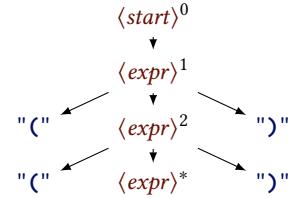
At this point, we will have a minimal input where the predicate reproduces the failure.

## 4 ABSTRACTION

*Abstraction* is the process of identifying the causative parts that contributed directly to the failure observed, and identifying and abstracting the non-causative parts. For abstraction, the idea is to identify which parts of the given derivation tree are required to produce a failure, and which parts can be replaced by a random generated string.

The algorithm is as follows. We start with the derivation tree that corresponds to the input string.

- (1) Identify the nonterminal of the top node of the tree.
- (2) Generate  $N$  random generated trees that correspond to the nonterminal where  $N$  is user configurable, and determined by the accuracy desired.



**Figure 6: The abstract derivation tree for  $((4))$ . The abstract nodes are marked with \*\*.**

- (3) For each generated tree, obtain the full input string from the full derivation tree where the current tree is replaced by the generated tree.
- (4) For each such string, check if predicate responds with FAIL. If for any generated tree, the predicate responds with PASS, then the tree cannot be abstracted. If all checked alternatives result in FAIL, we assume the tree can be abstracted. If predicate responds with UNRESOLVED, then the input failed to validate. We can only make the distinction between concrete and abstract using semantically valid inputs. Hence, we ignore the input—that is, we do not consider this input among the  $N$  random inputs, and generate a new input to try again.
- (5) If the tree can be abstracted, add the path to this node to *abstract nodes*.
- (6) If the tree cannot be abstracted, then continue the same procedure with its nonterminal children.

Using our example derivation tree at Fig. 5, we first consider abstracting  $\langle start \rangle^0$ . The generated strings for  $\langle start \rangle^0$  contain strings such as 100, 2 + 3 etc. These result in PASS from the predicate. That is,  $\langle start \rangle^0$  cannot be abstracted, and hence, marked as *concrete*. Similarly,  $\langle expr \rangle^1$ , also cannot abstracted, and marked as *concrete*. Considering  $\langle expr \rangle^2$ , all generated strings produced are parenthesized. However, only a few are double parenthesized. Hence,  $\langle expr \rangle^2$  is also marked as *concrete*. Considering  $\langle expr \rangle^3$ , all generated strings produced are of the form  $/((\cdot\cdot\cdot))/$  which successfully triggers the predicate. Hence, the path to  $\langle expr \rangle^3$  is added to *abstract nodes*. This results in the abstract derivation tree in Fig. 6.

This sequence of steps produces an annotated derivation tree where each node is marked as either abstract or concrete. This abstract derivation tree encodes what exactly caused the failure. In our case, the abstract derivation tree generates the string  $((\langle expr \rangle))$ , which clearly suggests the reason for failure—doubled parentheses.

## 5 ISOLATING INDEPENDENT CAUSES

Our abstraction algorithm introduced so far can fail when the failure is caused by interaction of multiple syntactical elements. For example, here is a minimized input string [7] for the *closure* JavaScript compiler: `{ while ((l_0)) { if ((l_0)) { break;; var l_0; continue }0 } }`. The problem here is that one requires exactly two instances of `l_0` to reproduce the bug with the remaining element allowed to be any syntactically valid identifier. So our first algorithm will try replacing the first `l_0`, and succeed because the other two instances are present in the string. Similarly, the second and third `l_0` will also be identified as abstractable because

the first and second respectively are present in the strings generated. However, the abstraction generated { while (((*Id*)) { if (((*Id*)) { break; ; var *Id*; continue }0 } } is incorrect – the failure would not be reproduced if all three are replaced by separate identifiers.

To address this problem, we verify that each of the nodes identified as abstract on its own, continued to do so when other abstract nodes are also replaced by randomly generated values. If the current node is no longer abstract when other abstract elements are replaced by randomly generated values, we unmark the current node as abstract, and run the abstraction algorithm on the child nodes, but with other abstract nodes replaced with random values.

The modified algorithm is given in Algorithm 2. The generate() function when given a derivation tree, and a set of paths, generates a new tree with all nodes pointed to by the paths replaced by a random compatible node.

We note that isolating independent causes is a well known problem for variants of delta debugging [15, 16], and our algorithm provides a solution if applied directly to non-reduced input (under the constraint that the faults can be isolated to separate parts of the input). That is, if the input to the algorithm contains two separate faults, both faults will be concretized and retained in the output.

## 6 IDENTIFYING SHARING ABSTRACTION

The abstraction algorithm detailed before works well when the examined parts are independent. However, complex languages often have elements such as variable definition and references that should be changed together.

As an example, Fig. 4a shows a failure-inducing input for the Rhino JavaScript interpreter [13]. There are a few questions that the developer may wish to answer here based on this fragment:

- (1) We see two empty {}. Do these need to be empty?
- (2) What exactly about the empty pattern is important here?
- (3) Can they be other data structures?
- (4) Is it required that the patterns are exactly the same?
- (5) Can one reproduce the failure with a variable other than baz?
- (6) Do the three baz need to be the exact same variable?

To further abstract such an input, we need to abstract over variable names—but in a *synchronized* fashion. To identify such patterns, we start with the abstract derivation tree, and first identify all non-terminals that are present in the derivation tree. Next, for each nonterminal, we identify its nodes in the derivation tree, and group the nodes based on the corresponding string representation. For any such grouping which contains more than one node, we generate compatible trees for that nonterminal and replace all the nodes in that group with that tree. We then obtain the string representation of the complete derivation tree and verify if the new string reproduces failure when the predicate is applied. If all such generated strings can reproduce the failure, the particular nodes under this grouping are parametrized by prefixing their nonterminal with a \$" and suffixing it with a unique id for this group.

For example, say we have the following input int v=0; v=1/v which induces an error (Fig. 7). The *abstraction* step will be unable to abstract v, and nodes annotated e, k and q and their parent nodes will be left concrete. Next, we look at the string representations of

---

### Algorithm 2 The abstraction and isolation algorithms

---

```

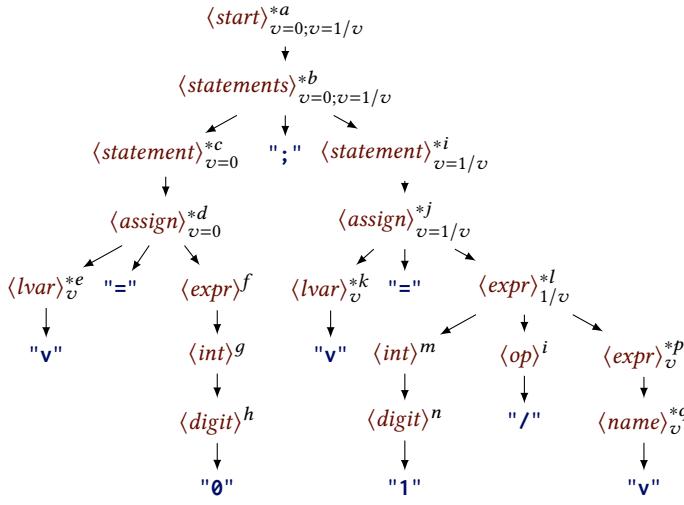
function CAN_GENERALIZE(tval, dtree, grammar, predicate, rnodes)
    checks  $\leftarrow$  0
    while checks < MAX_CHECKS do
        newtree  $\leftarrow$  generate(dtree, grammar, [tval] + rnodes)
        pres  $\leftarrow$  predicate(newtree.to_s)
        if pres = PASS then
            return false
        else
            if pres = FAIL then
                checks = checks + 1
            end if
        end if
    end while
    return true
end function

function ABSTRACTION(tval, dtree, grammar, predicate, rnodes)
    path, status  $\leftarrow$  tval
    if dtree.get(path).is_terminal then
        return []
    end if
    abstract  $\leftarrow$  can_generalize(tval, dtree, grammar, rnodes)
    if abstract then
        if status = UNCHECKED then
            return [(path, UNVERIFIED)]
        else
            return [(path, VERIFIED)]
        end if
    else
        paths  $\leftarrow$  []
        for child  $\in$  nonterminals(snode.children) do
            tval  $\leftarrow$  (child.path, UNCHECKED)
            paths.extend(abstraction(tval, dtree, predicate, rnodes))
        end for
        return paths
    end if
end function

function ISOLATION(tree, grammar, predicate)
    unv  $\leftarrow$  [((), UNCHECKED)]
    verified  $\leftarrow$  []
    original  $\leftarrow$  []
    while unv  $\neq$  [] do
        v  $\leftarrow$  unv.shift()
        notpv  $\leftarrow$  filter(original,  $\lambda o :!o.isparent(v)$ )
        topnodes  $\leftarrow$  filter(notpv,  $\lambda o :ischildofany(o, notpv)$ )
        runverified  $\leftarrow$  filter(unv,  $\lambda o :ischildofany(o, topnodes)$ )
        rnodes  $\leftarrow$  runverified + topnodes
        newpaths  $\leftarrow$  abstraction(v, tree, grammar, predicate, rnodes)
        for p  $\in$  newpaths do
            if p[1] == VERIFIED then
                verified.append(p)
            else
                unv.append(p)
            end if
        end for
        original.append(v)
    end while
    return mark_verified(tree, verified)
end function

```

---



**Figure 7: A derivation tree for `int v=0; v=1/v`. The concrete nodes are annotated with \* and the string representation is provided next to these nodes.**

each nonterminal. For example, the node  $d$  has a string representation  $v=0$  while  $k, e, p$ , and  $q$  has the same string representation  $v$ . We only collect string representations of concrete nodes (annotated by \* in Fig. 7). From the figure, we can see that the nodes  $a$  and  $b$  have similar string representation  $v=0; v=1/v$  similarly,  $c$  and  $d$  have  $v=0$ , and  $e, k, p$ , and  $q$  have  $v$  and so on. Out of these, we eliminate nodes that are a child of any of the other nodes in the grouping. This eliminates  $b$  since it is a child of  $a$ , which also eliminates the grouping since there is only one member left in the group.

This leaves us with one group, with members  $e, k$ , and  $p$  corresponding to  $v$ . Next, we pick a randomly generated value for one of the nodes; Say  $x$ , and use this value in place of each nodes, generating a new input string  $x=0; x=1/x$ . This string is now checked to see if it induces the same failure. (Only values that are legal to be replaced in each node is used, and values that result in UNRESOLVED are discarded). If the randomly generated value successfully reproduces the failure (as would happen in this case), we repeat the procedure for a fixed number of times for statistical confidence. If every such legal string induces the given failure, we mark the set of nodes as shared.

What if we have a set of nodes with the same string representation, but fails to vary together? E.g say we have an input `myval="myval"; check(myval)`. The input produces a failure if a variable with value "myval" is passed to the function `check()`. Here, `myval` can be replaced by any legal variable name so long it is correctly passed to `check()`. However, the string value has to remain "myval". To handle such cases, we produce combinations of nodes which are sorted by the number of nodes. That is, all nodes are checked for sharing first. Then, combinations with one node excluded are tried next, and then combinations with two nodes excluded and so on (denoted by `len_sorted_combinations()`). The first set of nodes that can vary together is chosen for a shared name. If no such set is found, the entire grouping is discarded. The complete algorithm is formalized in Algorithm 3.

---

### Algorithm 3 Identify shared nodes

---

```

function IDENTIFY_SHARED_NODES(tree, grammar, predicate)
    cpaths  $\Leftarrow$  concrete_paths(tree)
    snodes  $\Leftarrow$  find_similar(tree, cpaths)
    mtree  $\Leftarrow$  tree
    for key  $\in$  snodes do
        plst  $\Leftarrow$  snodes[key]
        paths  $\Leftarrow$  find_shared(plst, grammar, mtree, predicate)
        if paths then
            mtree = mark_context_sensitive(paths, mtree)
        end if
    end for
    return mtree
end function

function CONCRETE_PATHS(node)
    if node.abstract then
        return []
    end if
    my_paths  $\Leftarrow$  [node.path]
    for cnode  $\in$  node.children do
        my_paths = my_paths + concrete_paths(cnode)
    end for
    return my_paths
end function

function FIND_SIMILAR(tree, paths)
    strings  $\Leftarrow$  {}
    for path  $\in$  paths do
        node  $\Leftarrow$  tree.get(path)
        s  $\Leftarrow$  node.to_s
        if s.len = 0 then
            continue
        end if
        strings[(node.key, s)].add(path)
    end for
    res  $\Leftarrow$  filter(strings,  $\lambda s : \text{strings}[s].\text{len}() > 1$ )
    return res
end function

function FIND_SHARED(plst, grammar, tree, predicate)
    for paths  $\in$  len_sorted_combinations(plst) do
        checked = 0
        while checked < MAX_CHECKS do
            val  $\Leftarrow$  generate(tree, paths, same = true)
            res  $\Leftarrow$  predicate(val)
            if res = PASS then
                break
            else
                if res = FAIL then
                    checks = checks + 1
                end if
            end if
        end while
        if checked == MAX_CHECKS then
            return paths
        end if
    end for
    return {}
end function

```

---

The result of this step is the abstract failure-inducing input shown in Fig. 4b. With this, the questions that we asked above are immediately answered. First, we avoid ambiguity. The abstract failure-inducing input clearly indicates that all that is required is for the first two *(Id)* to be the same identifier (baz in the minimized string) and the first {} to be present.

Second, the abstract failure-inducing input suggests that the token *(variableDeclaration)* is effectively a space filler, as any instantiation of *(variableDeclaration)* would do. It further indicates precisely what portion of the processing program contributed to the failure—something related to the processing of the syntactical elements *(Id)* and *(variableDeclaration)*. This is an important information which was not visible in the minimized input fragment. In fact, this addresses a major drawback of current HDD variants as pointed out by Regehr et al. [16] (*generalized transformations*) and Pike [15] (*sharing*).

## 7 HANDLING LEXICAL TOKENS

In our last step, we tackle the distinction between *lexical* and *syntactical* features. Input strings often contain lexical parts such as whitespace and comments. These are often skipped over before parsing. These elements hence do not contribute to the program semantics. Showing that there is an abstract whitespace in between any two elements hence is of little help to the developer who interprets the abstract failure-inducing input.

Further, the abstract failure-inducing input may contain nonterminal symbols that represent optional parts of the grammar. These can be parts such as type hints in Python, documentation annotations, optional arguments to command lines that can be entirely skipped etc. While these are important sources of variability when the abstract failure-inducing input is used as a producer, a developer may not care about the existence of these nonterminals. Hence, we categorize them as *invisible* elements, and remove them from the compact representation of the abstract failure-inducing input.

Finally, tokens such as *begin* and *if* may be represented by nonterminal symbols such as *(BEGIN)* and *(IF)*. Since marking them as abstract does not provide any value-add for the developers, we identify such lexical tokens and replace them with their values in the compact representation.

## 8 THE COMPLETE DDSET ALGORITHM

Algorithm 4 shows how all the components fit together. The initial input is first parsed using a grammar, which results in a derivation tree. This derivation tree is passed to the reduction function, which minimizes the derivation tree to a *1-minimal-tree* input. The *1-minimal-tree* is then passed to the isolation function. The isolation function in turn, uses abstraction to identify nodes that are independently abstract. Finally, the function *identify\_shared\_nodes* identifies and marks parts that are shared. The tree thus produced is converted to a string representation and returned.

## 9 PROPERTIES AND LIMITATIONS

Our DDSET approach has a number of interesting properties and limitations, which we list here.

---

### Algorithm 4 Top level

---

```
function GET_ABSTRACTION(grammar, myinput, predicate)
    mtree  $\Leftarrow$  parse(myinput, grammar)
    rtree  $\Leftarrow$  reduction(mtree, grammar, predicate)
    itree  $\Leftarrow$  isolation(rtree, grammar, predicate)
    ctree  $\Leftarrow$  identify_shared_nodes(itree, grammar, predicate)
    return compact_rep(ctree)
end function
```

---

**Approximation.** In general, an abstract failure-inducing input will be an approximation. The reason is that a context-free grammar cannot fully characterize a universal grammar (that is, a Turing machine). There are specific causes for approximation:

- (1) During abstraction, if we are unable to generate any semantically valid inputs, we give up, and mark the node as concrete. This does not mean that the node cannot be generalized. Further, there may be other syntactical patterns that produce the same bug, which is not captured in the particular abstract failure-inducing input we derive from the minimized input string. Both of these are causes for *underapproximation*.
- (2) We have a fundamental limitation in that we rely on randomness to generate possible substitutes for particular nodes, and it is possible that the random strategy was unable to provide a counterexample in the time budget allotted. In such cases, we may erroneously mark nodes as abstract when they are not and *overapproximate*. This risk can be limited by running more tests.

**Limitations of Delta Debugging.** Our approach inherits limitations from delta debugging, such as reliance on a precise test case [16] that may require knowledge about internal program structure (such as a crash location). We also require that the test case clearly distinguishes between FAIL and UNRESOLVED, especially as several of the generated test inputs may be valid syntactically, but invalid semantically. Finally, we require that the test case be deterministic—that is, the program behavior fully depends on the given input.

**Grammar Quality.** Like Perseus and other HDD variants, DDSET relies on a grammar to reduce and abstract inputs. Since it uses the grammar both for decomposing as well as for producing inputs, it is important to have a high-quality grammar. If the grammar is too lax, it will parse inputs well; many ANTLR parsers err on this side. However, many of the *produced* inputs will be invalid, requiring the test to strictly separate between FAIL and UNRESOLVED. If the grammar is too tight, opportunities for generalization will be missed.

**Performance.** We note that DDSET is computationally expensive. In particular, in the worst case, each node in the derivation tree contributes to  $N$  executions of the test case where  $N$  is dependent on the desired accuracy. The number of nodes in a derivation tree depends on the branching factor, and can be approximated to  $O(n \log(n))$  where  $n$  is the number of tokens, and the base of the logarithm is the branching factor. Furthermore, DDSET may need to generate a number of syntactically inputs for generating a single semantically

valid input. If it required  $K$  syntactically valid inputs for one semantically valid input, the worst case runtime complexity of DDSET would be  $O(K \times N \times n \times \log(n))$  where  $n$  is the number of tokens in the input. The number of tests, however, is in line with Delta Debugging on complex inputs; as DDSET would typically be started automatically after an automated test has failed, there is no human cost involved.

**At least as good as Delta Debugging.** Despite the above limitations, let us point out that DDSET is at least as good as the state of the art. Notably, its *abstract failure-inducing inputs are never longer than the reduced failure-inducing input*. We always start with the minimized input, and abstraction only substitutes character sequences with single tokens. Hence, the length of the result (counting each token as a single element) will never be more than the length of the original minimized input. If there exists a better variant for HDD than Perves, we can simply use that variant instead.

## 10 EVALUATION

To evaluate DDSET, we pose the following research questions.

- (1) **RQ1** How effective is the DDSET algorithm in generating abstract failure-inducing inputs? That is, we want to know if DDSET algorithm can accurately identify abstractable patterns in the given input, and whether identifying these patterns can lead to an overall reduction in the complexity of the input.
- (2) **RQ2** How accurate are the patterns generated by the DDSET algorithm? That is, did the algorithm correctly identify parts that can be abstracted? Or were some parts mislabelled as abstract?

### 10.1 Evaluation Setup

For our evaluation, we used subjects for four input languages, from programming languages to command lines:

**Javascript** is a large language with numerous parser rules, keywords, and other special context rules. We used the Javascript grammar definition from the ANTLR project [2], which corresponds to the ECMAScript 6 standard. The ANTLR Javascript grammar contains both lexical (lexer) and syntactical (parser) specification. We used the following Javascript interpreters in the evaluation:

- Closure interpreter v20151216, v20200101, and v20171203. The bugs were obtained from the Closure issues page [8].
- Rhino interpreter version 1.7.7.2. The bugs for this project were obtained from the Github issues page [14].

**Clojure**<sup>5</sup> is a Lisp-like language with a limited set of parse rules that describe the main language. We used the Clojure grammar definition published by the ANTLR project [1] in 2014, which still parses later versions of Clojure. The following version was used for evaluation:

- Clojure interpreter version: 1.10.1. The bugs for this project were obtained from Clojure JIRA [6].

**Lua**<sup>6</sup> is a smaller language with a limited set parsing rules. We used the Lua grammar definition from the ANTLR project [3]. The following version was used for evaluation:

- Lua interpreter 5.3.5. The bugs for this project were obtained from the project page [11].

**UNIX command line utilities.** For the UNIX command line utilities, we chose the two commands *find* and *grep* which were published in the *DBGbench* [4] benchmark. The grammars for *find* and *grep* command line options were extracted from the manual pages. These grammars list which parts of the command line are optional, and which parts accept arguments. The syntax for arguments is also represented and includes regular expressions and file names. The particular coreutil bugs are identified by their hashes in *DBGbench*.

We converted each grammar from the ANTLR format to a pure context free form by extracting *optional* and *star* patterns to separate grammar rules. For each bug, we read the bug report, and identified the *smallest* input that was provided in the bug report by the reporter or later commenter. Using this input, we translated the bug behavior to a test case with the following properties:

- Successfully identify when the fault is triggered (FAIL).
- For complex languages (those except coreutils) the test case should identify whether the semantic rules of the language were fulfilled (PASS).
- Similarly, for complex languages, the test case should be able to accurately identify when the semantic rules are violated, leading to a rejection of the input (UNRESOLVED). This is because, for languages such as Clojure, Javascript and Lua, there is often a second stage after parsing where the program is statically analyzed to identify errors. Hence, while according to the grammar of Clojure, any parenthesized expression can be placed anywhere, a developer normally expects a specific kind of expression under, say, a parameter definition. For coreutils, the UNRESOLVED indication was not used.
- Triggering the timeout (one minute) was counted as PASS.

### 10.2 RQ1: Effectiveness of Abstraction

The effectiveness of abstraction aims to capture how effective DDSET is in identifying non-essential filler parts. To measure the effectiveness of abstraction, we assess the number of nonterminal symbols that could be used in a given minimal string. This indicates the amount of abstraction possible in that a nonterminal can be replaced by any of its semantics conforming expansion. The remaining characters in the string indicates what could not be abstracted in this way, and indicates the limit of DDSET.

Our results for abstraction are shown in Table 1:

- The **Bug** column contains the bug identifier in the particular bug tracking system the language uses. “rhino 385”, for instance, denotes the bug in Fig. 4a.
- The **# Chars in Min String** column reports the length of the input string after it was minimized through the *peres* reduction algorithm. This is our starting point.
- The **# Visible Nonterminals** column reports the number of distinct nonterminals found by the abstraction algorithm that are visible to the user in the abstract failure-inducing input— the more nonterminal symbols found, the better the abstraction was, and the lower the cognitive load for the developer. In the pattern for “rhino 385”, shown in Fig. 4b, we have three visible nonterminals.

Note that the Nonterminal count does not include number of nonterminal symbols for space, comments, and other lexically skipped parts of the program that are not part of the semantics. Any space left after minimization is counted as part of remaining characters. Secondly, any abstractable element that resulted in an empty string is skipped.

- The **# Invisible Nonterminals** column denotes the number of nonterminal symbols that are invisible (because they represent empty string or skipped space and comments). We see that the abstract failure-inducing input in Fig. 4b has 17 invisible nonterminals, denoting the spaces between elements.
- Within the nonterminals, the number of shared nonterminals is provided in the column **# Shared**. An item such as  $2 + 3$  indicates that two distinct shared nonterminals were found, of which the first had a two references, while the second had three references—the more such shared symbols found, the better (even better than nonterminals found) as it indicates areas of semantic importance, and hence, possible places that the developer should focus on. For Fig. 4b, this item is 2, which represents the single shared nonterminal (`<Id1>`) which appears twice. For *lua-5.3.5.4*, this is  $3 + 2$  which indicates one shared nonterminal that repeats thrice, and another that repeats two times.
- The **# Remaining Chars** column reports the number of characters left after removing the tokens—for Fig. 4b, these are 7.
- Finally, the **# Executions** column reports the number of executions required for the complete abstraction (including minimization). It takes 14,015 executions of Rhino to obtain the abstract failure-inducing input in Fig. 4b. We note that the large number of executions is due to the high accuracy desired, and the accuracy desired can be controlled by the user.

In total, Table 1 shows that all 22 bugs could be abstracted to varying degrees. In particular, nine bugs had a number of shared elements which should get extra attention from the developer. Any of the abstract failure-inducing inputs stands for an infinite set of inputs, which can all be used to generate more tests. Finally, the table shows that DDSET never does worse than the *minimized input* it started with.

DDSET could provide abstract failure-inducing inputs for all 22 bugs studied.

While the number of executions may seem high, it is in line with expectations (See “Performance” in Section 9) and the state of the art. It is also high because we used up to 100 test runs to validate each abstraction step. Reducing this number to, say, 10, would much increase performance, but also increase the risk of inaccuracy.

Users can choose between high accuracy and a lower number of executions.

**Table 1: The effectiveness of abstraction on reported bugs**

Bug	# Chars in Min String	# Visible Nonterminals	# Invisible Nonterminals	# Shared	# Remaining Chars	# Executions
lua-5.3.5.4	83	5	54	$3+2=5$	28	19265
clj-2092	55	1	24	=0	11	505
clj-2345	34	1	12	=0	4	911
clj-2450	185	5	60	$2=2$	27	1954
clj-2473	29	2	18	=0	10	5118
clj-2518	30	0	20	=0	8	741
clj-2521	135	2	23	=0	10	864
closure 1978	84	10	37	$3=3$	11	17174
closure 2808	14	3	7	$2=2$	1	167
closure 2842	60	6	41	$3+3=6$	14	551
closure 2937	35	2	21	=0	7	8203
closure 3178	42	7	26	$2=2$	8	14853
closure 3379	16	0	11	=0	4	935
rhino 385	33	3	17	$2=2$	7	14015
rhino 386	16	2	13	$2=2$	6	557
grep 3c3bdace	37	1	2	=0	31	236
grep 54d55bba	64	1	1	=0	61	239
grep 9c45c193	21	2	1	$2=2$	19	117
find 07b941b1	16	1	4	=0	15	280
find 93623752	11	1	3	=0	10	192
find c8491c11	15	1	3	=0	14	200
find dbcb10e9	15	2	3	=0	13	381

### 10.3 RQ2: Accuracy of Abstraction

The abstractions provided by DDSET are only useful if developers can be confident that they are *accurate*—that is, they are correct abstractions for a multitude of possible instantiations. Hence, we also judge the accuracy of our abstractions by generating random inputs from the produced abstract failure-inducing input, and checking how effective it was in generating failure-triggering inputs. Given that we are evaluating the effectiveness of the abstract failure-inducing input as a producer, we do not have to worry about the cognitive load of the developer. Hence, *invisible* nonterminals (i.e optional elements) are allowed during production.

For generating strings, we simply chose a random expansion for each of the abstract patterns. The complete string thus produced is evaluated using the test case.

The invalid values are produced because while generalizing, we filtered out produced invalid values from a given nonterminal. This means any nonterminals we use can produce invalid values. For example, consider the Closure bug 2937. The minimized string is `var A = class extends (class {}){}{}` and the abstraction produced is `var <assignable> = class extends (class<classTail>){}{}`. According to the Javascript grammar, *<assignable>* can be any *<objectLiteral>*,

**Table 2: Test generation from abstracted patterns**

Bug	VALID %	FAIL %	Bug	VALID %	FAIL %
clj-2092	100	100	closure 3379	76	100
clj-2345	100	100	lua-5.3.5 4	100	100
clj-2450	62	100	rhino 385	49	100
clj-2473	40	100	rhino 386	100	100
clj-2518	100	100	grep 3c3bdace	100	100
clj-2521	100	100	grep 54d55bba	100	100
closure 1978	76	100	grep 9c45c193	100	100
closure 2808	100	100	find 07b941b1	100	100
closure 2842	100	99	find 93623752	100	100
closure 2937	36	100	find dbcb10e9	100	100
closure 3178	57	100	find c8491c11	100	100

which through a long chain of rules allows `<singleExpression>`, which allows almost all other parts of Javascript to be present. However, only a limited number of items are allowed in the `<assignable>` position. This validation is handled external to the parser. Unfortunately, given that this validation is absent for the fuzzer, it produces inputs of the type: `{var { T[yield] (){return}} = class extends (class {}){}}` where the fragment `{ T[yield] (){return}}` is allowed by the grammar, but invalid as an `<assignable>` according to the language semantics.

The results for accuracy are given in Table 2. While not all inputs produced are valid, almost all valid inputs derived are actually failing. The term “abstract failure-inducing input” thus is adequate. This is especially true for simple languages such as `find` and `grep` command lines, which perform very well in terms of accuracy.

*Instantiating abstract failure-inducing inputs produced by DDSET produces failures with high accuracy. On average, 86.5% of inputs produced were valid, of which 99.9% succeeded in reproducing the failure.*

## 11 THREATS TO VALIDITY

Our evaluation has the following threats to validity:

**External Validity.** The external validity (generalizability of our results) of our results depends on how representative our data set is. Our case study was conducted on a small set of programming language interpreters and UNIX utilities. Further, we evaluate only a limited number of bugs for their abstractability. Hence, there exist a threat that our samples may not be representative of the real world. A mitigating factor is that these were real bugs logged by real people, and the grammars we used are from some of the most popular and complex programming languages that are used to build real world applications. Hence, we believe that our approach can be generalized to other subjects, especially subjects with less complex languages.

**Internal Validity.** The threat to internal validity refers to the correctness of our implementation and evaluation. We mitigated this by verifying that all our algorithms and programs work on a small set of well understood examples before applying

it to our sample set of bugs. Every resulting abstract failure-inducing input is simple enough to quickly spot errors.

**Construct Validity.** The threat to our construct validity (are we measuring what we claim to be measuring) is how useful the effectiveness and accuracy measures are. Short of a user study, this threat cannot be completely mitigated. However, we note that our abstraction intuitively models how a developer thinks about a fault causing input, i.e. “here is a variable, and if I use this particular variable in this fashion later, it triggers a bug”. We also note that simplification and generalization seem to be qualities whose usefulness for users is universally accepted.

## 12 RELATED WORK

Despite their importance for debugging, the study of failure-inducing inputs and their characteristics is very limited. Notably, the question of determining *sets* of failure-inducing inputs has, to the best of our knowledge, never been addressed in the literature; DDSET thus opens the door to a new field of research.

Generalizing failure-inducing inputs has been mostly studied in the context of *reducing* them. The original algorithm for *delta debugging* for program reduction was introduced by Zeller and Hildebrandt [19]. Delta debugging works by partitioning the input sequence into chunks and checking whether smaller and smaller chunks can be discarded while retaining the required property.

A number of variants for delta debugging exist. The first research to target structured inputs was HDD [12] by Misherghi and Su. HDD was motivated by finding that delta debugging performs poorly on structured inputs. HDD applies delta debugging on a single level of the hierarchy at any given time. Herfert et al [10] introduced tree transformations for hierarchical reduction and showed that in typical programming languages, parent nodes could often be replaced by one of the child nodes, leading to better reduction. Perses [18] by Sun et al. uses an *input grammar* to guide reduction. The innovation of Perses is in realizing that one could use the node type and look for similar node types in the descendant nodes for replacement. The other innovation is in identifying that certain kinds of nodes can be entirely deleted if the nonterminal definition has an empty rule. DDSET uses Perses as its initial reduction step and generalizes from there, using the input grammar also to produce additional test inputs to validate abstractions.

Reduction algorithms can also be specialized for individual input languages. ChipperJ [17] uses Java specific transformations to produce a minimal cause preserving input. A similar research is C-Reduce by Regehr et al [16] for C programs. Similar to ChipperJ, it applies a sequence of valid reducing transformations on C code to obtain a minimal cause preserving input. ChipperJ and C-Reduce achieves minimization through their in depth semantic knowledge of the Java and C programs, and together outperform other more general variants of hierarchical delta debugging. The identification of context-sensitive nodes, as implemented in DDSET, specifically targets identifier definition and usage in programming languages.

Among the specializations of delta debugging for specific domains, Bruno’s SIMP tool [5] for reducing failure-inducing SQL queries stands out in that it attempts to distinguish fixed (necessary) parts in the failure-inducing input from variable parts. In

contrast to DDSET, SIMP does not attempt to generalize inputs to a maximum or to produce an abstract failure-inducing input.

Another notable approach close to our own is *universal sub-value generalization* by Lee Pike [15]. The approach by Pike shows how one can use a *forall* construct in *Haskell* to indicate generalizable constructors in the *Haskell* representation of an AST. However, compared to our approach, Pike's approach (1) does not generate a compact representation, (2) does not understand how to deal with lexical elements, (3) produces *unsound* generalization when independent sub-causes exist, and (4) does not identify context-sensitive parts such as defined variables (sharing).

Persees and DDSET require a grammar for their reduction, abstraction, and testing steps. Such grammars would typically be specified manually, which can be quite some effort. Recent work on mining grammars from dynamic control flow [9] suggests that such grammars can also be extracted from programs by dynamically tracking individual input characters. In our setting, such a learner could be applied to produce a grammar from the failing program and its failure-inducing input, requiring no further specification effort.

### 13 CONCLUSION AND FUTURE WORK

What are the inputs that cause a failure? We present the first algorithm that not only reduces a given failure-inducing input to a short string, but also *generalizes* it to a set of related failure-inducing inputs. This abstract failure-inducing input is short and easy to communicate, and represents the set of failure-inducing inputs with high accuracy. By producing sets rather than single inputs, DDSET has a clear advantage over classical reduction algorithms.

In our future work, we want to take the generalizations of DDSET further—notably by systematically exploring the surroundings of the original input, not only reducing it, but also by adding additional elements in order to determine the context under which a failure takes place or not. Eventually, we thus want to be able to determine an entire *language* that not only serves as a producer of failure-inducing inputs, but also as a *recognizer*, being able to predict whether a given input will cause a failure or not. We expect several usage scenarios for such a recognizer, notably as it comes to detecting malicious inputs.

DDSET and all evaluation data is available as open source at

<https://github.com/vrthra/ddset>

### ACKNOWLEDGMENTS

Andreas Zeller's team at CISPA provided helpful comments on earlier revisions of this paper. We also thank the anonymous reviewers for their highly constructive comments.

### REFERENCES

- [1] Antlr. 2020. ANTLR Clojure Grammar. <https://github.com/antlr/grammars-v4/tree/master/clojure>. Online; accessed 27 January 2020.
- [2] Antlr. 2020. ANTLR Javascript Grammar. <https://github.com/antlr/grammars-v4/tree/master/javascript/javascript>. Online; accessed 27 January 2020.
- [3] Antlr. 2020. ANTLR Lua Grammar. <https://github.com/antlr/grammars-v4/tree/master/lua>. Online; accessed 27 January 2020.
- [4] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2014)*. 105–115.
- [5] Nicolas Bruno. 2010. Minimizing Database Repros Using Language Grammars. In *Proceedings of the 13th International Conference on Extending Database Technology (Lausanne, Switzerland) (EDBT '10)*. Association for Computing Machinery, New York, NY, USA, 382–0393. <https://doi.org/10.1145/1739041.1739088>
- [6] Clojure. 2020. Clojure Bugs. <https://clojure.atlassian.net/secure/Dashboard.jspa>. Online; accessed 27 January 2020.
- [7] Google. 2020. Closure 2842 Bugs. <https://github.com/google/closure-compiler/issues/2842>. Online; accessed 27 January 2020.
- [8] Google. 2020. Closure Bugs. <https://github.com/google/closure-compiler/issues>. Online; accessed 27 January 2020.
- [9] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2020*.
- [10] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-structured Test Inputs. In *IEEE/ACM Automated Software Engineering (Urbana-Champaign, IL, USA) (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 861–871. <http://dl.acm.org/citation.cfm?id=3155562.3155669>
- [11] Lua. 2020. Lua Bugs. <https://www.lua.org/bugs.html>. Online; accessed 27 January 2020.
- [12] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *International Conference on Software Engineering*. 142–151.
- [13] Mozilla. 2020. Rhino 385. <https://github.com.mozilla/rhino/issues/385>. Online; accessed 27 January 2020.
- [14] Mozilla. 2020. Rhino Bugs. <https://github.com.mozilla/rhino/issues/385>. Online; accessed 27 January 2020.
- [15] Lee Pike. 2014. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4–5, 2014*, Wouter Swierstra (Ed.). ACM, 53–64. <https://doi.org/10.1145/2633357.2633365>
- [16] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.
- [17] Chad D Sterling and Ronald A Olsson. 2007. Automated bug isolation via program chipping. *Software: Practice and Experience* 37, 10 (2007), 1061–1086.
- [18] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Persees: Syntax-guided Program Reduction. In *International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. ACM, New York, NY, USA, 361–371. <https://doi.org/10.1145/3180155.3180236>
- [19] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 183–200. <https://doi.org/10.1109/32.988498>

# Debugging the Performance of Maven’s Test Isolation: Experience Report

Pengyu Nie

University of Texas at Austin, USA  
pynie@utexas.edu

Aleksandar Milicevic

Microsoft, USA  
almili@microsoft.com

Ahmet Celik

Facebook, Inc., USA  
celik@fb.com

Jonathan Bell

George Mason University, USA  
bellj@gmu.edu

Matthew Coley

George Mason University, USA  
mcoley2@gmu.edu

Milos Gligoric

University of Texas at Austin, USA  
gligoric@utexas.edu

## ABSTRACT

Testing is the most common approach used in industry for checking software correctness. Developers frequently practice reliable testing—executing individual tests in isolation from each other—to avoid test failures caused by test-order dependencies and shared state pollution (e.g., when tests mutate static fields). A common way of doing this is by running each test as a separate process. Unfortunately, this is known to introduce substantial overhead. This experience report describes our efforts to better understand the sources of this overhead and to create a system to confirm the minimal overhead possible. We found that different build systems use different mechanisms for communicating between these multiple processes, and that because of this design decision, running tests with some build systems could be faster than with others. Through this inquiry we discovered a significant performance bug in Apache Maven’s test running code, which slowed down test execution by on average 350 milliseconds *per-test* when compared to a competing build system, Ant. When used for testing real projects, this can result in a significant reduction in testing time. We submitted a patch for this bug which has been integrated into the Apache Maven build system, and describe our ongoing efforts to improve Maven’s test execution tooling.

## CCS CONCEPTS

- Software and its engineering → Software configuration management and version control systems.

## KEYWORDS

Build system, Maven, test isolation

### ACM Reference Format:

Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven’s Test Isolation: Experience Report. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395363.3397381>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA ’20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397381>

## 1 INTRODUCTION

Previous research shows that tests in industry are riddled with flakiness [8, 19, 20, 28, 30, 32]. A common practice to combat flaky tests is to run them in isolation from each other. This aims to eliminate *test-order dependencies*, i.e., potential effects that one test execution may have on executions of other tests (e.g., by polluting some shared static state) [6, 7, 23, 42]. Depending on the level of isolation, however, the introduced overhead can be substantial.

Test isolation can be implemented to various degrees, ranging from full sandboxing (i.e., each test is run in its own freshly provisioned OS virtual machine) to sequential execution of all tests inside a single OS-level process. The former approach provides absolute isolation, though at a very high cost; the latter, in contrast, introduces minimal overhead but only guards against races caused by concurrency.

Speaking of Java projects, running all tests inside a single Java Virtual Machine (JVM) is the default mode for many build systems, including Ant, Gradle, and Maven. Although efficient, using this default mode may impact the correctness of test results and lead to intermittent test failures when there exist test-order dependencies.

The most commonly used middle ground is running each test in its own process. In case of Java projects, this entails *forking* a new JVM for every test. The overhead of this technique (compared to running all tests in a single JVM) is known to be significant. Prior work has shown that the slowdown can be as high as several orders of magnitudes [6]. The high cost of forking is commonly associated with the cost of spawning a large number of JVM processes.

Recent work introduced VmVm [6], an approach that defines test virtualization to isolate each test within a single JVM. Specifically, VmVm tracks accesses to static fields during a test execution and automatically re-initializes those fields prior to the execution of the subsequent test. VmVm brings the best from the two worlds: test isolation via lightweight containers and reuse of all loaded classes among tests by executing everything in a single JVM. Extensive evaluation of VmVm showed significant performance improvements over forking and low overhead compared to running all tests inside a single JVM.

Despite its powerful approach, we believe that VmVm faces two key obstacles on its way to a wider adoption. It is a complicated system, which involves dynamic bytecode instrumentation, and such instrumentation must be updated with each new version of Java. Perhaps rightfully so, many developers may be hesitant to be early adopters of research tools, when developers need tools that they can rely on, but researchers are pressured to keep inventing new

tools over maintaining older work. Instead, developers rely widely on open source frameworks and tools that are already adopted and maintained by the community.

Inspired by the initial VmVm evaluation and aforementioned challenges, we set out to seek a simple yet effective approach to optimize forking, with the goal of creating a solution that we could ensure would be more widely adopted. We began by performing an *exploratory study* to compare the performance overhead of test isolation in three different build systems (Ant, Maven and Gradle). Although all three of these build systems executed the exact same JUnit tests, each build system provides a separate implementation of the glue that actually *executes* those tests. In this study, we evaluated the overhead of executing a trivial test, which simply called `Thread.sleep(250)`, repeatedly, in new processes. Somewhat surprisingly, our results of profiling build system runs show that, for Maven in particular, a large portion of the overhead is unexplainable by OS-imposed overheads: on average each test took approximately 350 milliseconds longer to run with Maven than with Ant or Gradle.

To better understand and isolate the sources of overhead in test isolation, we created FORKSCRIPT, a highly-optimized JUnit test runner that runs each test in its own process. During a build, on-the-fly, FORKSCRIPT generates a single *specialized execution script* for running all tests and collecting test results. Our execution script utilizes process management provided by the operating system, as well as several utilities available on Unix systems. The build system, hence, needs to interact only with that execution script process, instead of all individual test processes.

Of course, FORKSCRIPT could suffer the same adoption limitations as VmVm: FORKSCRIPT needs to be tightly integrated with the build system and test framework and represents a maintenance challenge. Hence, developers may choose to continue to allow the build system to run their tests as it normally would, and again, we may not be successful in getting developers to adopt FORKSCRIPT. Instead, we used FORKSCRIPT as a baseline while we performed a deep-dive into Maven's test execution performance, and were able to reveal a bug in the inter-process communication (IPC) code that is used for test communication in Maven. This bug imposed a constant overhead of approximately 350 milliseconds per-test-class, resulting in a particularly enormous relative overhead in projects with very fast running tests. Our patch for this bug has been integrated into Maven, resulting in a significant speedup for test runs that require test isolation. Moreover, since our contribution is merged into Maven (and is not a standalone tool or Maven plugin), we can be confident that it will have practical impact.

This experience report describes our process discovering, debugging and patching this bug, as well as our experiences working with the community to continue to improve Maven's IPC code. Our results show that FORKSCRIPT can save up to 75% (50% on average) of test execution time compared to Maven forking when running a single JVM at a time. The benefits of FORKSCRIPT for runs with parallel processes slightly decrease, although savings remain high. Most importantly, our results also show that with our patch that has been merged into Maven, every developer can see similar performance to FORKSCRIPT. Reflecting on the experience of developing VmVm, FORKSCRIPT, and then patching Maven itself, it is clear to us that working with open source communities can be both extremely challenging and rewarding. Developing FORKSCRIPT

was far easier than debugging and patching the underlying flaw in Maven. Although FORKSCRIPT ultimately was somewhat faster than our patched version of Maven, by creating and integrating a simple patch into Maven we have finally been able to have much broader impacts by getting our improvements in front of every developer who uses Maven.

The main contributions of this paper include:

- A closer look at the cost of forking (as implemented in the existing build systems) and a finding that substantial overhead behind it comes from generic inter process communication (IPC).
- A novel, simple, technique, dubbed FORKSCRIPT, to optimize forking by creating specialized test execution script on-the-fly.
- A deep-dive performance measurement of test execution in Ant, Maven and Gradle, resulting in a patch that has been merged into Maven.
- An extensive evaluation of our approach on 29 open-source projects, totaling 2 million lines of code.
- Impact on several design decisions for future releases of the Maven build system [3, 5].

Our experience report clearly shows that researchers should explore rigorous and systematic testing of build systems, including their performance, correctness, and usability. These topics have not received much attention by the testing community in the past.

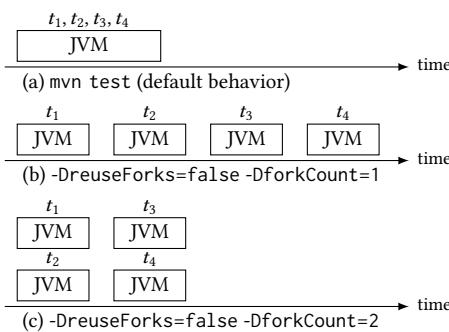
## 2 MOTIVATING STUDY

While past work on test acceleration typically focuses on a *single* build system (e.g. all of the projects evaluated in the VmVm paper used the Ant build system [6]), we were particularly curious if different build systems' approach to test isolation could result in different performance. Hence, we conducted a simple experiment to compare the overhead that each build system adds when executing tests with isolation. To do so, we generated a test suite consisting of 100 JUnit test classes, each with a single JUnit test method that simply calls `Thread.sleep(250)` and then returns. These simple tests are not appropriate for benchmarking complex runtime systems like VmVm, but provide the perfect opportunity to measure overheads introduced by the build system, since each test should require a constant execution time. We measured the total time taken by each of Ant, Maven and Gradle to execute this entire test suite (with test isolation, and excluding the time taken by other parts of the build process). Then, we subtracted the actual time needed to run each test (250 msec) to identify the overhead per-test introduced by each build system.

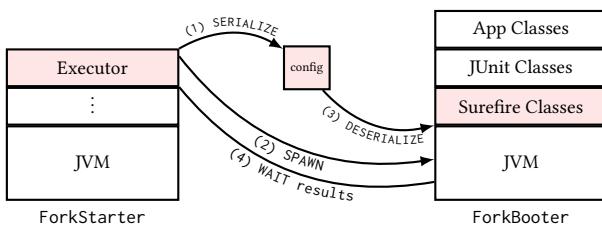
Table 1 presents the results of this study. We found that the overhead introduced by each build system varied dramatically, from just 259 msec in the case of Ant to 596 msec in the case of Maven. This wide range makes us hopeful that there may be things that can be changed in Maven or Gradle to reduce the overhead of test isolation to at least that of Ant. We know that there will have to be *some* overhead to running tests in this method, since there is a non-zero overhead imposed by the operating system when creating a new process. However, *all* of the build systems will have to pay for that same overhead, and hence, any differences in overhead between build systems would have to be explained by

**Table 1: Overhead Introduced by Each Build System for Isolating a Single Test in Milliseconds.**

Build System	Overhead Per-Test (ms)
Ant 1.10.6	259
Gradle 5.6.1	412
Maven (Surefire 3.0.0-M3)	596



**Figure 1: Illustrative example of three Surefire configurations when running four tests ( $\{t_1, t_2, t_3, t_4\}$ ).**



**Figure 2: Surefire’s workflow. Shaded boxes highlight the main sources of overhead introduced by Surefire.**

different IPC mechanisms. Hence, if we can identify why Maven is so much slower than Ant, we could, ideally propose a change to Maven to improve its performance. Such a change could afford developers with some of the speedup that they might obtain from using a system like VmVm to isolate their tests, but with the added convenience of not requiring *any* per-project changes, by making a change to the build system directly. At the same time, while we can see from this data that Ant has the lowest overhead of these three options, we don’t know for certain that it represents an ideal solution: perhaps an even more efficient system could be faster.

### 3 BACKGROUND

We provide a high level overview of Maven and describe the key steps taken by Maven when executing a set of tests.

Maven defines a build lifecycle—a well-defined sequence of *phases*, including compilation, code generation, test execution, etc. A user can specify a set of plugins to be executed with each build phase; a plugin is associated with a phase by its developers, and a user can only choose which plugins to include in the build run. An official plugin for running unit tests is Maven Surefire [3]. Surefire

**Input:** tests - set of tests

**Input:** config - build configuration

```

1: function FORKSTARTER(tests, config)
2:   executor ← MAKEEXECUTOR(config.forkCount)
3:   for τ in tests do:
4:     classpath ← GETCLASSPATH(τ, config)
5:     serializedConfig ← SERIALIZED(τ, classpath)
6:     task ← MAKECOMMANDLINETASK(serializedConfig)
7:     SUBMIT(executor, task)
8:   done
9:   WAITFORALL(executor)
10: end function

```

**Figure 3: Overview of the key steps executed in the ForkStarter class, which is a part of the Surefire plugin; this code is executed in the same Java Virtual Machine that is running the main Maven process.**

implements complex logic for finding test classes in a project, filtering tests, etc. A rich set of options enables user to fine tune test execution [4].

To execute tests in a project, a developer can simply execute the following command: `mvn test`. This command starts Maven (which is written in Java and executes in its own JVM), goes through all phases until (and including) the test phase and runs user specified plugins; all the plugins are executed in the same JVM as the main Maven process, although the plugins can spawn other processes. The aforementioned command would trigger Surefire plugin to spawn *a single* new JVM process and execute *all tests* in the new JVM (Figure 1.a). If a user wants to isolate tests, by running each test in a separate JVM, she can use the following command: `mvn test -DreuseForks=false -DforkCount=1` (Figure 1.b). The first command-line option (`reuseForks`) specifies that JVM should not be reused after test execution is finished, and the second command-line option (`forkCount`) specifies the number of JVM instances that should run in parallel. Figure 1.c illustrates a configuration with two JVMs running in parallel.

Behind the scenes, the Surefire plugin finds the tests, extracts the build configuration needed to tune the test execution, and prepares options for a new JVM. We only consider execution *with* forking (without reusing JVMs) in the remainder of this section, i.e., configurations illustrated in Figure 1.b and Figure 1.c. When it comes to actual test execution, there are two key classes involved: ForkStarter and ForkBooter. Figure 2 visualizes their interaction. ForkStarter executes in the same JVM as Maven and spawns new JVMs. ForkBooter is the main class started in the new JVM process, which reads configuration prepared by ForkStarter and executes one test.

Figure 3 summarizes the key steps that are executed in the ForkStarter class. The input to ForkStarter is (1) the set of tests to execute, and (2) a parsed build configuration. Initially (line 2), ForkStarter creates an executor, i.e., an instance that maintains a thread pool [2], where each thread will be responsible for spawning new JVMs and triggering test executions. In the next step (lines 3-8), ForkStarter iterates over the set of tests to create one task for each test and submit the task to the executor. Each iteration of the loop makes a classpath for the current test, serializes the configuration

```
Input: serializedConfig - path to serialized configuration file
1: function FORKBOOTER(serializedConfig)
2:   config ← DESERIALIZECONFIG(serializedConfig)
3:   SETUPJVM(config)
4:   junit ← MAKEJUNIT
5:   EXECUTE(junit, config.test)
6:   SENDGOODBYE()
7: end function
```

**Figure 4:** Overview of the key steps executed in the ForkBooter class, which is a part of the Surefire plugin; this code is executed in the new Java Virtual Machine.

for the new JVM to a temporary file, creates a task, and submits the task. Once all tasks are submitted, ForkStarter waits for all tasks to complete (line 9).

Figure 4 summarizes the key steps that are executed in the ForkBooter class, which is the main class in newly spawned JVM. In the first step (line 2) ForkBooter deserializes the configuration from the temporary file, sets up configuration for the current JVM (line 3), creates an instance of JUnit (line 4), and executes the test with the JUnit (line 5). Finally (line 6), it sends a goodbye signal (via standard output) to the ForkStarter.

#### 4 BEST-CASE PERFORMANCE: NO IPC

When isolating test case executions in their own process, the test running infrastructure must have some form of *interprocess communication* (IPC) to coordinate between the two processes. We speculated that this must be where the increased overhead that we saw from Maven and Gradle in comparison to Ant. Based on our initial profiling of Maven runs and considering the steps taken, in ForkStarter and ForkBooter, to execute a set of tests, we highlight, in Figure 2 the steps that introduce overhead *in addition to* the cost of spawning new JVM processes. Specifically, (1) using thread pool and executors to manage processes requires additional class loading and adds substantial complexity especially for those test runs when only a single JVM is run at a time (i.e., there is a single thread in the thread pool), (2) exchanging configuration with new JVMs via serialization/deserialization to/from files requires costly IO operations, (3) class loading of Surefire’s classes (e.g., ForkBooter) in each new JVM adds on top of already costly class loading of classes under test [33], and (4) “pumping” input/output between the original JVM and newly created JVMs requires extra threads, synchronization, etc.

Although some of the extra steps taken by Surefire may be necessary for certain build configurations, we believe that the overly generic design adds substantial overhead for simple configurations, which are common for many small and medium sized open-source projects, as those used in our evaluation. Thus we set to design and develop a simple but effective approach to remove overly generic steps, taken by Surefire, whenever such approach is feasible.

To reduce the IPC between a build system and the processes it spawns, we present FORKSCRIPT. During a build, on-the-fly, FORKSCRIPT generates a single specialized execution script for running all configured tests and collecting test results; the tests may be run sequentially or in parallel. Our execution script utilizes process

```
Input: tests - set of tests
Input: config - build configuration
Output: S - specialized script that executes tests
1: function FORKSCRIPT(tests, config)
2:   count ← 0
3:   for τ in tests do:
4:     classpath ← GETCLASSPATH(τ, config)
5:     S+ ← $timeout ‘config.timeout \$’
6:     S+ ← $java -cp ‘classpath
7:           forkscript.JUnitRunner ‘τ ‘config &$
8:     if (count + 1) % config.forkCount == 0 then:
9:       S+ ← $wait$
10:      fi
11:      count++
12:   done
13: end function
```

**Figure 5:** A simplified version of a multi-staged program used by FORKSCRIPT to generate a specialized script; this program replaces ForkStarter code in Figure 3.

management provided by the operating system, as well as several popular utilities available on Unix systems. The build system, hence, needs to interact only with that shell script process, instead of all individual test processes.

FORKSCRIPT replaces ForkStarter and completely removes ForkBooter. At the same time, other parts of Maven and Surefire need not be changed at all. Moreover, we rely on Surefire to discover tests, perform filtering, extract build configuration from configuration files, etc. The changes introduced by FORKSCRIPT happen when it comes to concrete test execution steps.

Figure 5 shows a simplified multi-staged program [37], which replaces code in Figure 3 and Figure 4, used by FORKSCRIPT to generate a specialized script for the given set of tests and build configuration. We use standard notation from code generation community to represent code fragments and holes [26].  $\leftarrow^+$  concatenates the strings and appends a newline character.

The input to the FORKSCRIPT function is the same as for ForkStarter: a set of tests and the build configuration. The FORKSCRIPT function, similar to ForkStarter, iterates over each test in the set of tests, but rather than adding each test as a task to an executor, FORKSCRIPT extracts configuration (e.g., classpath for the test) and appends a shell command that will execute the test with the appropriate setup. Depending on the number of JVMs that should run in parallel (forkCount), FORKSCRIPT splits the set of tests in one or more *buckets*. Note that tests within each bucket are executed in parallel, as scheduled by the OS, but each test runs in its own JVM. In our implementation, we use the split command (available on Unix) to split tests into buckets.

To execute each test, FORKSCRIPT uses a custom JUnitRunner, which, unlike the default JUnit runner (org.junit.runner.JUnitCore), accepts several command line arguments to fine tune the output of the runs. Clearly, to be able to support other testing frameworks (e.g., TestNG) or a different version of the same framework, we would have to implement one custom runner for each testing framework and each version.

Because the specialized script is generated in each test run, FORKSCRIPT does not introduce any burden on the developers to maintain those scripts. In other words, changes in the set of tests (if tests are added or deleted), as well as changes in the classpath, are automatically reflected in the subsequent test runs. This transparency for users enabled an easy transition to FORKSCRIPT.

We believe that a large number of projects with a simple build configuration, can substantially benefit from FORKSCRIPT. This is also evidenced by the results of our evaluation (see Section 6). Considering the frequency of test runs, especially for those projects that use Continuous Integration [15, 25, 39], e.g., Travis CI, FORKSCRIPT can have substantial impact on developers’ productivity and software reliability.

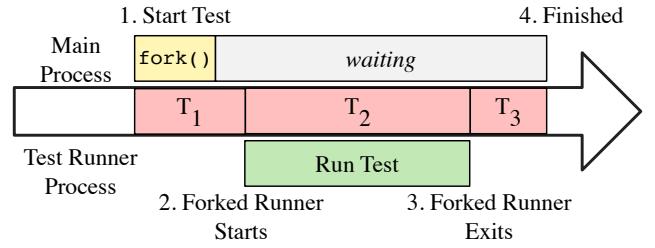
## 5 PERFORMANCE PROFILING MAVEN

FORKSCRIPT provides a barebones, stripped down mechanism to run JUnit test classes in their own process, but may not suit all developers’ needs. Hence, we *also* carefully profiled Maven to identify the specific source of the ~350msec/test performance overhead (in comparison to Ant). To start, we refined our overhead measurement experiment described in Section 2 to break down the time measurement into three components. In particular, we wanted to attribute the overhead into time needed to launch a new process, for that process to run the test, and then for that process to terminate. Figure 6 shows an abstract timeline of the execution of a single JUnit test class which is representative for Ant, Maven and Gradle. We profiled:  $T_1$ , the duration between when the build system begins running a test until the child process (the test runner) starts;  $T_2$ , the duration between when the child process starts until the child process terminates; and  $T_3$ , the duration between when the child process terminates until when the build system determines that the test has completed.

To conduct this profiling, we implemented a simple load-time Java bytecode instrumentation agent that modified the appropriate methods of each build system to record the system time of each of the four events shown in Figure 6. Again, we ran our profiler on 100 tests that each simply call `Thread.sleep(250)`, and report the average duration of each of the steps to execute each test in Table 2. We were surprised to find that the majority of Maven’s overhead (compared to Ant) came from  $T_3$ , the duration between when the test ends (in the ForkedBooter child process) to when the parent process (the build system, aka the ForkStarter) finds that the test has completed.

With this clue in hand, we attached a Java debugger to the child process (ForkedBooter) and set a breakpoint at the point in the execution that represents the start of  $T_3$  in Figure 6 – when the test completes. We examined the program behavior carefully, and found that the program was stalling while a thread was reading from the standard input stream, `<stdin>`. Furthermore, we found that it was this thread blocking on `<stdin>` that was causing a roughly 350 millisecond delay for each test execution.

Upon further investigation, we found that this is not a problem unique to Maven. Normally, when a Java thread is reading from an `InputStream`, it can be interrupted by another thread (for instance, if the JVM is shutting down). However, when reading from `<stdin>`, the thread can *not* be interrupted until they reach a JVM “safepoint”, which occurs every 300 milliseconds [38].



**Figure 6: Critical regions in test execution to profile.** We focus on three regions: 1) between when the build system decides to execute a test and when the forked process launches, 2) between when the test starts and when it finishes, and 3) between when the test finishes and when the build system considers the test complete.

**Table 2: Profiling Results that Break Down the Time to Run Each Test Using the Durations Described in Figure 6. Our Patch Significantly Reduces the  $T_3$  Measurement.**

Build System	$T_1[\text{ms}]$	$T_2[\text{ms}]$	$T_3[\text{ms}]$
Ant 1.10.6	250	253	9
Gradle 5.6.1	395	253	17
Maven (Surefire 3.0.0-M3)	244	253	352
Maven (With our patch)	217	252	17

To resolve this bug, the fix is simple: when the ForkedBooter (child JVM) determines that it is time to shut down, do not continue to read from `<stdin>`. We proposed this three-line patch (plus a one-line change to an integration test) to the Maven Surefire developers, who gladly accepted and merged it into their master branch [14]. With this change, we shaved approximately 350 milliseconds off of *each* test class execution time for Maven, bringing its performance much closer in line with Ant. Since the change was integrated directly into the project, every user of Apache Maven stands to benefit from this improvement. We are interested in applying a similarly detailed performance analysis to Gradle’s performance in the future.

Interesting future work would also consider the implications of using network sockets over standard input for test running communication. In addition to avoiding OS peculiarities regarding blocking on `<stdin>` versus other streams, migrating communication to sockets would allow Maven to run tests on separate physical machines than the machine invoking Maven – a powerful feature. We began working with the Apache Maven community on developing such a feature, but development is still underway: the test running code in Maven is extremely old, and it was not designed with the goal of making such drastic IPC changes easy.

## 6 STUDY

To assess the benefits of our contributions, we answer the following research questions:

**RQ1:** What are the performance improvements obtained by FORKSCRIPT compared to the default Maven forking?

**RQ2:** How does the improvement scale as the number of concurrent processes increase?

**RQ3:** How does the patched Maven compare to FORKSCRIPT?

We run all experiments on a 4-core Intel Core i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 18.04. We used Oracle Java HotSpot(TM) 64-Bit Server (1.8.0\_181). Furthermore, we used Maven 3.5.3.

We first describe the subjects used in our study, as well as experiment setup, then we answer our research questions.

## 6.1 Subjects

To select the subjects for our study, we mostly searched through recent work on regression testing and test isolation [6, 21, 23, 27]. Additionally, we set a couple of requirements for each project: the project must (1) be buildable with the Maven build system, (2) have non-trivial number of tests, (3) have tests whose execution time is non-negligible, and (4) successfully build at its latest revision. We use the latest revision of each project available at the time of (the latest run of) our experiments rather than revisions used in prior studies; this makes our experiments feasible as building old revisions can be challenging.

Table 3 shows the list of 29 projects used in our experiments. All the selected projects are open-source and available on GitHub. For each project, we show the size of the project in terms of the number of lines of code (LOC); number of Maven modules (#Modules); number of source files in the repository (#Files); number of test classes, i.e., tests (Classes); number of test methods (Methods); location of the project on GitHub (URL); and the revision used in our experiments (SHA). We used cloc [13] to collect LOC, our scripts to collect number of files, and Maven default output to report the number of tests and test methods.

Figure 7 shows the distribution of test execution time, per project, for all tests; the x-axis is in the  $\log_{10}$  scale.

Although projects in Table 3 may not be representative of all Java projects (or even Java projects that build with Maven), we believe that the set of chosen projects covers a broad range of application domains. Additionally, we can see that the projects vary in LOC, number of files, number of tests, and number of test methods. The last two rows in Table 3 (Avg. and  $\Sigma$ ) show the average and total values (where applicable).

## 6.2 Experiment Setup

We briefly describe our experiment setup. The main goal is to compare test execution time if one uses Maven forking vs. FORKSCRIPT. To that end, we perform the following changes and steps.

We modify recent version of Surefire to output the exact time taken for the entire test execution. Note that Surefire by default prints only time for *each test* and the total execution time for the *entire build*, but it does not print the total time for test execution as defined by the ForkStarter function in Figure 3. Thus, we modified ForkStarter to output time at the beginning and at the end; this is the only change in Surefire for our experiments. We did the same change for the FORKSCRIPT function in Figure 5. As FORKSCRIPT has no impact on other phases of the Maven build life cycle, we believe that measuring only test execution time provides a fair comparison and avoids any potential noise from other phases in the build.

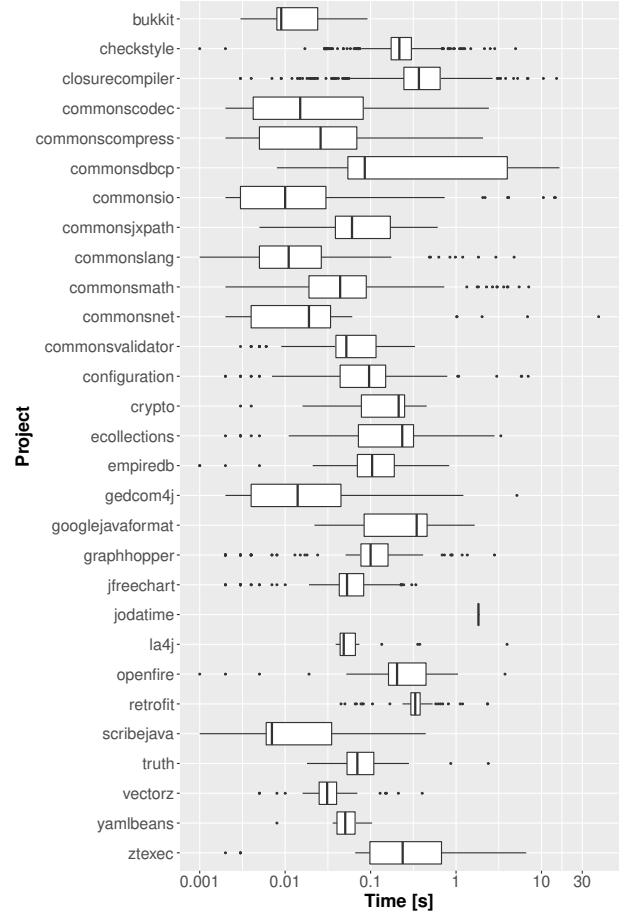


Figure 7: Distribution of test execution times in  $\log_{10}$  scale.

For each project in Table 3 we execute the following steps:

- Clone the project from GitHub and checkout the revision shown in Table 3 (SHA),
- Install the project (by executing `mvn install`), which will execute all the build phases and download necessary dependencies (i.e., third party libraries),
- Run tests using Maven in the offline mode to avoid unnecessary connection to the Internet, and
- Run tests using FORKSCRIPT in the offline mode.

We store the logs for the last two steps to be able to extract the number of tests executed in each run, as well as the test execution time. We use the former to check the correctness of our runs, i.e., each FORKSCRIPT run should execute the same number of tests as the Maven run. We manually confirmed that all build outcomes are the same. We use the latter to compute the savings of FORKSCRIPT compared to default Maven implementation. Namely, we extract total test execution time for each Maven run (denoted with  $T^{\text{Mvn}}$ ) and FORKSCRIPT run (denoted with  $T^{\text{FS}}$ ) and compute savings as  $RT = \frac{T^{\text{Mvn}} - T^{\text{FS}}}{T^{\text{Mvn}}} \times 100\%$ . This saving is the key metric used in our evaluation. We use the same metric for both sequential runs and parallel runs.

**Table 3: List of Subjects Used in our Study.**

Project	LOC	#Modules	#Files	#Tests		URL https://github.com/	SHA
				Classes	Methods		
bukkit	32,560	1	731	38	903	Bukkit/Bukkit	f210234
checkstyle	206,965	1	2,001	297	2,993	checkstyle/checkstyle	f7626ac
closurecompiler	357,610	3	1,196	354	13,713	google/closure-compiler	5cae9b7
commonscodec	20,400	1	127	54	869	apache/commons-codec	1406889
commonscompress	44,711	1	354	137	1,057	apache/commons-compress	e127b13
commonsdhcp	20,654	1	106	30	569	apache/commons-dhcp	5226462
commonsio	30,455	1	248	105	1,354	apache/commons-io	9e5475f
commonsjxpath	24,910	1	232	38	409	apache/commons-jxpath	3138e7a
commonslang	76,203	1	325	148	4,084	apache/commons-lang	58a8f12
commonsmath	153,695	1	1,177	367	4,158	apache/commons-math	b05b3b9
commonsnet	28,207	1	272	43	268	apache/commons-net	cc819eb
commonsvalidator	16,082	1	149	70	541	apache/commons-validator	a377131
configuration	66,847	1	457	169	2,786	apache/commons-configuration	bc69f94
crypto	5,964	1	87	25	111	apache/commons-crypto	d69ef95
eclipsecollections	298,762	16	2,578	3,196	161,417	eclipse/eclipse-collections	f2543f2
empiredb	52,700	7	472	25	108	apache/empire-db	f816837
gedcom4j	47,204	1	503	243	1,300	frizbog/gedcom4j	be310f2
googlejavaformat	13,951	1	69	17	1,037	google/google-java-format	579772a
graphhopper	59,435	7	565	137	1,460	graphhopper/graphhopper	b2db5c2
jfreechart	133,847	1	987	341	2,176	jfree/jfreechart	520a4be
jodatime	86,138	1	330	1	4,222	JodaOrg/joda-time	c9f2764
la4j	13,581	1	117	20	835	vkostyukov/la4j	2826a4a
openfire	203,070	8	1,589	31	235	igniterealtime/Openfire	6c32335
retrofit	19,339	5	220	65	595	square/retrofit	5c2f505
scribejava	11,160	6	238	25	109	scribejava/scribejava	09364b6
truth	28,805	2	179	59	1,447	google/truth	14f72f7
vectorz	53,487	1	397	72	456	mikera/vectorz	a05c69d
yamlbeans	6,716	1	51	8	89	EsotericSoftware/yamlbeans	f561099
ztxec	3,163	1	76	20	93	zeroturnaround/zt-exec	6c3b93b
Avg.	72,986	2	545	211	7,220	N/A	N/A
$\Sigma$	2,116,621	75	15,833	6,135	209,394	N/A	N/A

### 6.3 Improvements For Sequential Runs

*RQ1: What are the performance improvements obtained by FORKSCRIPT compared to the default Maven forking?*

Overall, we find that savings obtained by FORKSCRIPT for sequential runs are 50% on average and up to 75%. We discuss the details below.

To answer this research question, we followed the experiment setup from the previous section and executed tests with the following command: `mvn test -DreuseForks=false -DforkCount=1`. Recall that the first command-line option will enforce that each test is executed in its own JVM, and the second option specifies that there is only one forked JVM at a time. In other words, we are executing tests sequentially one JVM at a time. Recall (Section 4) that FORKSCRIPT too respects the `forkCount` parameter, so in this case FORKSCRIPT too will run a single test at a time in its own JVM. We use `Fork1` to denote this run configuration (for both Maven and FORKSCRIPT); we do not encode `reuseForks` in the name of

the configuration, because *all* our experiments use this option, i.e., enforce full test isolation.

Table 4 shows the results. The first column lists names of subjects, and columns 6 and 7 show test execution time in seconds for Maven and FORKSCRIPT, respectively. Column 8 shows savings (RT) of FORKSCRIPT compared to Maven. Interestingly, we can observe that FORKSCRIPT, for `Fork1`, speeds up test execution for all subjects. Minimum speedup is 12% for `commonsdhcp` and maximum speedup is 75% for `bukkit`. The last two rows show the average and total values for all projects.

We took an extra step to check what projects benefit the most from FORKSCRIPT. We mostly reasoned about the obtained speedup, number of tests (as we already reported in Table 3), and test execution time per test (as we already showed in Figure 7). We observed that projects with low median time per test benefit the most; median is shown as the vertical line for each boxplot in Figure 7. This is not surprising, as the overhead of forking is more observable for tests that run shorter. For example, FORKSCRIPT was more beneficial for

**Table 4: Test Time in Seconds of Maven and FORKSCRIPT, with NoFork and Fork with 1 and 2 Parallel Processes. RT is the Time Savings of FORKSCRIPT Compared to Maven.**

Project	NoFork				Fork 1				Fork 2			
	$T^{mvn}$ [s]	$T^{FS}$ [s]	RT[%]	$T^{new}$ [s]	$T^{mvn}$ [s]	$T^{FS}$ [s]	RT[%]	$T^{new}$ [s]	$T^{mvn}$ [s]	$T^{FS}$ [s]	RT[%]	$T^{new}$ [s]
bukkit	0.98	0.51	47	0.79	19.07	4.62	75	6.79	7.01	2.56	63	4.18
checkstyle	28.48	27.86	2	28.82	231.64	129.27	44	146.17	114.62	82.03	28	90.45
closurecompiler	68.07	74.12	-8	71.11	408.43	289.74	29	301.97	233.35	209.39	10	210.08
commonscodec	6.86	6.40	6	6.62	34.65	13.11	62	16.77	14.44	9.27	35	9.75
commonscompress	11.21	10.14	9	11.11	82.75	29.40	64	40.93	33.56	17.11	49	22.23
commonsdhcp	70.53	70.28	0	73.49	90.01	78.59	12	80.43	49.91	42.15	15	47.45
commonsio	58.45	57.80	1	58.13	107.79	69.70	35	75.18	52.56	45.94	12	38.60
commonsjxpath	1.88	1.57	16	1.64	21.14	8.48	59	9.33	8.74	4.75	45	5.30
commonslang	13.49	13.38	0	13.50	88.15	32.11	63	41.51	34.56	22.29	35	22.81
commonsmath	53.12	53.18	0	53.49	243.92	109.10	55	243.62	103.24	61.21	40	103.28
commonsnet	57.51	58.47	-1	57.93	78.93	63.16	19	65.77	48.46	52.02	-7	47.82
commonsvalidator	1.86	1.60	13	1.74	37.25	12.27	67	15.86	13.80	7.76	43	9.25
configuration	24.85	24.14	2	24.14	127.49	61.50	51	70.60	59.38	34.89	41	40.10
crypto	1.86	1.50	19	1.59	15.48	7.17	53	8.45	7.15	4.40	38	5.04
eclipsecollections	76.81	71.00	7	76.13	2,233.72	1,157.09	48	1,307.81	1,014.09	670.55	33	737.34
empiredb	2.36	1.68	28	2.06	14.48	6.38	55	6.70	6.29	4.17	33	4.36
gedcom4j	17.05	9.35	45	18.44	149.74	46.24	69	77.05	64.77	28.65	55	46.01
googlejavaformat	4.29	3.81	11	4.18	15.67	9.90	36	11.46	8.05	6.84	15	7.28
graphhopper	20.18	14.46	28	19.46	91.34	41.33	54	50.71	43.33	27.98	35	31.50
jfreechart	2.10	1.40	33	2.14	174.61	59.41	65	77.70	62.74	32.32	48	43.85
jodatime	2.79	2.26	18	2.68	2.99	2.28	23	2.79	2.92	2.48	15	2.73
la4j	5.77	5.15	10	5.62	15.25	7.70	49	9.35	7.48	5.51	26	6.39
openfire	8.61	7.80	9	7.66	29.73	16.77	43	16.71	13.85	12.02	13	10.87
retrofit	21.15	17.31	18	17.35	53.24	35.67	33	36.22	30.83	26.41	14	24.89
scribejava	4.19	2.46	41	2.85	13.61	4.23	68	5.88	6.65	3.35	49	4.05
truth	5.65	4.93	12	5.05	35.99	14.57	59	17.39	16.37	9.15	44	10.76
vectorz	1.49	1.01	32	1.21	37.92	11.00	70	14.87	14.58	6.21	57	8.50
yamlbeans	0.70	0.32	54	0.43	4.24	1.20	71	1.66	1.69	0.73	56	1.09
zexec	15.58	15.46	0	15.94	25.93	19.61	24	20.86	14.39	14.56	-1	12.10
<b>Avg.</b>	20.27	19.28	15	20.18	154.66	80.74	50	95.88	72.02	49.88	32	55.45
<b><math>\Sigma</math></b>	587.87	559.35	N/A	585.30	4,485.16	2,341.60	N/A	2,780.54	2,088.81	1,446.70	N/A	1,608.06

bukkit than for closurecompiler (75% vs. 29%). As another example, FORKSCRIPT was more beneficial for commonscodec than for commonsdhcp (62% vs. 12%). An exception to this rule is commonsnet, which has low median value and relatively low saving (19%). With a closer look, we found that a few tests in commonsnet dominate the entire test execution time and the number of tests in this project is small, thus the cumulative overhead is small and savings are low.

For completeness, we also report test execution time without test isolation. Namely, we simply execute: mvn test with Maven and FORKSCRIPT. The results are shown in Table 4 in columns 2-4 (NoFork). Although FORKSCRIPT provides savings even in this case, we note that the absolute difference in test execution time is rather small, and this configuration was not the motivation for our work.

#### 6.4 Improvements For Parallel Runs

*RQ2: How does the improvement scale as the number of concurrent processes increase?*

Overall, our results show that FORKSCRIPT outperforms Maven for parallel runs. Concretely, FORKSCRIPT saves, on average, 32% of test execution time when running 2 JVMs in parallel. Next, we discuss the results in details.

To execute tests in parallel we execute the following command: mvn test -DreuseForks=false -DforkCount=2. We have also obtained results with  $-DforkCount = 4$ , which showed similar results, but we do not report the details in this paper. As our machine has 4 cores, we have not tried running experiments with larger number of parallel processes.

Table 4 shows the results for parallel runs. Columns 10-12 (Fork2) show execution time and savings when running (up to) two JVMs in parallel. We can see that savings remain substantial. On average, for runs with two parallel processes, FORKSCRIPT saves 32%. For runs with two parallel processes, minimum saving (slowdown in this case) is -7% while maximum is 63%. The reduction in savings, when running multiple JVMs in parallel, was expected as the total execution time approaches theoretical maximum, i.e., time to

execute the longest test in the project. Note that we expect that runs for jodatime are the same for all forking options because this project has only one test (although it has many test methods).

## 6.5 Comparison With Patched Maven

*RQ3: How does the patched Maven compare to FORKSCRIPT?*

Finally, we compared FORKSCRIPT with the patched Maven. Clearly, we expected that FORKSCRIPT would outperform the patched Maven, but we expected to see much smaller differences compared to the non-patched Maven version.

Table 4 shows the results. In columns titled “ $T^{new}$ ” we report time for executing tests with the patched Maven. We can observe that savings of patched Maven are substantial over the non-patched version. Additionally, FORKSCRIPT slightly outperforms patched Maven, as was expected (Section 4), because FORKSCRIPT supports only a subset of features.

## 7 IMPLICATIONS AND LESSONS LEARNED

Reflecting on our experience, we believe that there are several important lessons that we have learned which are applicable to practitioners and researchers:

**Detect performance bugs through differential testing.** To the best of our ability to judge, this performance bug has been in Maven since the introduction of the feature to isolate tests (beyond the history of the project’s git repository). Developers everywhere have likely assumed that the performance bug that we found was simply the normal, expected behavior. Only by performing fine-grained differential testing of three different build systems (described in Section 2) did we even recognize that there was a flaw. Performance bugs are notoriously difficult to find, but when there are alternative systems that accomplish the same goal, differential testing can help to reveal them.

**Find simple fixes that can be integrated today.** While our fix solved the performance bug that we found, it could also be solved by a long-term effort that is rewriting the entire ForkedBooter and ForkStarter IPC system that relies on `<stdin>`. This architecture dates back to the creation of this project (over a decade ago), and unfortunately poses problems for projects that, themselves use `<stdin>` and `<stdout>`. Instead, there is a significant refactoring within the Maven community to replace the `<stdin>` communication with non-blocking socket channels [1]. However, this effort will take several months, and require the involvement of multiple Apache community members. When we first isolated the bug, we actually went down this route *first* as a fix, and almost prepared a pull request to make this change in Maven. While we made it work (and pass all tests), the changes required to the codebase were extremely invasive. We created the simple patch described in Section 5 only after realizing that it would take months to transition the project from `<stdin>` to sockets in a way that would best comply with the community practices.

**For researchers: engage in the open source community.** As software testing researchers, engaging with the open source community is an extremely valuable opportunity to both discover interesting problems and to have a broader impact on the community. Open source projects can have a diverse group of contributors, each of whom may have another software development job, with

their own differing perspectives and challenges. Engagement can come in many rich forms: in the past we have reported bugs and made pull requests to Maven, but never before had we joined the Slack channel. We encourage other researchers to engage with the open source community not only by finding and reporting bugs in projects, but by patching them and integrating new features that the community wants. Compared to our past experiences contributing to the same project (Maven), we found that it is far easier to have a pull request merged when our change is either to fix a clear bug or to implement a feature that already is on developer’s wishlists. In contrast, proposing pull requests that simply integrate our research prototypes or research ideas into the open source project are far less likely to be successful, since open source project maintainers might be wary to accept large new features that may not be necessary in the eyes of the maintainer, but nonetheless will become a maintenance burden for that person. Finding a balance between research novelty and practical impact is key, as always. In the meantime, we continue to work with the project maintainers to migrate the IPC from `<stdin>` to socket channels, and look forward to continue to contribute to Maven.

**For researchers: perform systematic testing of build systems.** There is a vast amount of literature related to build systems, but it mostly focuses on build system design (e.g., [24]) and testing correctness of build scripts (e.g., [9]). Researchers have invested limited effort in checking correctness of build systems, with an exception of several efforts to formally design and prove (a part of) build systems [12, 31]. This experience report clearly shows the need for more work in this direction.

## 8 THREATS TO VALIDITY

**External.** The set of projects used in our evaluation may not be representative of all Java projects. To mitigate this threat, we used a large number of open-source projects, which differ in size, application domain, and number of tests. Moreover, many of the projects used in our study have been used in prior work on (regression) testing [6, 11, 21, 27].

The reported results are obtained on a single machine. However, we obtained results for many projects across four different machines, and the savings were substantial in all cases. We do not report results from all machines as showing those numbers would not add new insights.

We used a single revision of each project in this paper, and the results could differ for other revisions. We simply chose the latest revision of each project available at the time of our experiments; in our initial experiments, not reported in this paper, we used earlier revisions, and we observed similar savings when using FORKSCRIPT.

**Internal.** Our code and scripts may contain bugs that could impact our conclusions. To increase our confidence in our implementation, we did many sanity checks. Specifically, we checked that the number of tests is the same regardless of the build configuration and we confirmed that each log file contains the format appropriate for the used configuration. In addition, we wrote tests for our code and did pair programming.

**Construct.** We have not compared our approach with other techniques that have been developed to speed up execution of tests, including regression test selection [35, 41], test case minimization [34],

and efficient class loading [33]. We believe that those other techniques are *orthogonal* to our approach and could provide additional benefits if used together. We discuss the details of other techniques in the Related Work section.

FORKSCRIPT and VmVm share the same overall goal (reducing time needed to run tests in isolation), but take radically different approaches. VmVm requires bytecode instrumentation to change each test to be self-isolated, and is quite brittle and can result in unexpected test failures. It was also not designed for Java 8 (but Java 7), let alone today’s Java 14. Instead, FORKSCRIPT changes the build system to run each test more efficiently. We were faced with 1) using only very old, Java 7, and few projects for our evaluation, 2) use an apparently updated version of VmVm that is also fragile and unsupported (<https://github.com/Programming-Systems-Lab/vmvm/issues/7>), or 3) not report any unfair comparison in this paper. We thought that choice 3 was the fairest for everyone.

## 9 RELATED WORK

**Test-order dependencies and test isolation.** Muslu et al. [32] showed that running tests without isolation hides bugs. Zhang et al. [42] were among the first to demonstrate the impact of test order dependence on test outcome. They also proposed a technique to detect test order dependencies and showed the impact of detected dependencies on five test-case prioritization techniques. Bell and Kaiser [6] performed an extensive study to evaluate the cost of isolating tests via forking and presented a technique, named VmVm, for unit test virtualization. Later work by Bell et al. [7] presented ElectricTest, an approach for detecting dependencies among tests by utilizing JVM’s garbage collection and profiling. By forcing garbage collection at the end of each test, ElectricTest captures if objects modified by one test are read by the subsequent test(s). Gyori et al. [23] introduced PolDet to detect tests that pollute shared state. PolDet captures the state of the heap at the beginning and at the end of each test and reports if the captured states differ. Unlike prior work, we focused on identifying hidden overhead in forked runs, and we proposed a technique to remove costly IPC for those build configurations when such communication is not necessary.

**Flaky tests.** Test-order dependency is only one source of flaky tests, i.e., tests that non-deterministically pass or fail for the same input without any change to the program [19, 20, 28, 30]. Other sources of flakiness include non-determinism due to thread scheduling, network access, IO, etc. Initial work on flaky tests mostly focused on creating taxonomy of flaky tests and avoiding flakiness. Recent work focuses on automatically detecting flaky tests [8]. Our work is on avoiding (rather than detecting) flaky tests due to test dependencies.

**Build systems.** A build system is in charge of orchestrating the execution of build tasks [16, 24, 29, 36]. Although most build systems support a handful of built-in in process tasks, in practice most build tasks involve spawning a process and waiting for it to complete. Such orchestration, no matter how simple it may be, inevitably introduces some overhead. Depending on the number of advanced features the build system may support (e.g., incrementality, shared caches, process sand-boxing) [12, 17, 22], the introduced overhead may be significant. For example, to implement a simple timestamp-based incrementality, the build system has to record all input and

output files of a build task and check their timestamps before it determines if the task needs to be rerun; to support fetching build outputs from a shared cache, a fingerprint of the build task must be determined, which (among other things) includes computing the checksums of all input files; finally, if the goal is to ensure that a build task doesn’t read/write any files other than those it declares, the task must be run in a sandbox or otherwise monitored for file accesses it makes.

All those advanced features carry certain benefits at the cost of adding more overhead in the worst-case scenario (e.g., clean builds with cold caches). It is up to the developers of a project to decide what is the best trade-off for building their project. From the build system’s point of view, FORKSCRIPT replaces a number of fine-grained build tasks (individual test executions) with a single build task (which executes all tests). Making the granularity of build tasks coarser is likely to undermine the benefits of the aforementioned advanced features (incrementality/cacheability/sandboxing). However, many projects are not set up to use those features in the first place (or their build system simply doesn’t support them), in which case there is no penalty to pay for using FORKSCRIPT. Using FORKSCRIPT is also appropriate in continuous integration loops which are typically configured to run clean builds only [10, 25, 39].

**Other techniques to speed up JVM runs.** Nikolov et al. [33] proposed recoverable class loaders, an approach to snapshot class loaders and associated class objects. Xu and Rountev [40] presented an approach for identifying inappropriate use of data structures in Java programs. Nailgun [18], which is currently maintained by Facebook developers, provides a client, protocol, and server for running Java programs without paying the startup cost. Our work is orthogonal to prior efforts and optimizes test isolation by avoiding overly generic implementation of inter process communication.

## 10 CONCLUSION

This experience report captures our efforts to demystify and better understand why test isolation is computationally expensive. The result of our efforts – a patch integrated into the popular Apache Maven build system – will result in a significant reduction in test execution time for many developers. Based on these results, we also created a research prototype, FORKSCRIPT. FORKSCRIPT is based on our finding that a large portion of test execution time, if test isolation is needed, goes into overly generic infrastructure of build systems, e.g., inter process communication. To reduce this time, FORKSCRIPT creates a specialized script for a given build configuration and maintains the script as the build configuration changes. FORKSCRIPT is publicly available. Our approach of using differential testing of multiple build systems to detect performance bugs could be useful for other researchers or practitioners. We continue to engage with the open source community to further improve the inter process communication components in Maven.

## ACKNOWLEDGMENTS

The authors thank Karl Palmskog, Chenguang Zhu, and the anonymous reviewers for their comments and feedback. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517.

## REFERENCES

- [1] Apache. 2018. *Test XML file is not valid when rerun "fails" with an assumption.* <https://issues.apache.org/jira/projects/SUREFIRE/issues/SUREFIRE-1556>.
- [2] Apache. 2018. *Thread Pool in Maven Surefire Code.* <https://github.com/apache/maven-surefire>.
- [3] Apache. 2019. *Maven Surefire Plugin.* <https://maven.apache.org/surefire/maven-surefire-plugin/>.
- [4] Apache. 2019. *Maven Surefire Plugin — surefire:test.* <https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>.
- [5] Apache. 2019. *Should Surefire specialize test runner when test isolation (i.e., fork) is needed?* <https://issues.apache.org/jira/browse/SUREFIRE-1516>.
- [6] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *International Conference on Software Engineering*. 550–561.
- [7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *International Symposium on Foundations of Software Engineering*. 770–781.
- [8] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*. 433–444.
- [9] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Softw. Engg.* 22, 6 (2017), 3117–3148.
- [10] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *International Symposium on Foundations of Software Engineering*. 643–654.
- [11] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *International Symposium on Foundations of Software Engineering*. 809–820.
- [12] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *International Symposium on Formal Methods*. 643–657.
- [13] Al Danial. 2020. *Cloc.* <https://github.com/AlDanial/cloc>.
- [14] Tibor Digana. 2019. [SUREFIRE-1516] Poor performance in reuse-Forks=false. <https://github.com/apache/maven-surefire/commit/5148b02ba552cd79ac212b869dec10d01ba4d2e6>.
- [15] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.
- [16] Sebastian Erdweg, Moritz Lichter, and Weiel Manuel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Object-Oriented Programming, Systems, Languages & Applications*. 89–106.
- [17] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft’s Distributed and Caching Build Service. In *International Conference on Software Engineering, Software Engineering in Practice*. 11–20.
- [18] Facebook. 2020. *Nailgun.* <https://github.com/facebook/nailgun>.
- [19] Martin Fowler. 2018. *Eradicating Non-Determinism in Tests.* <http://martinfowler.com/articles/nonDeterminism.html>.
- [20] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should We Control?. In *International Conference on Software Engineering*. 55–65.
- [21] Milos Gligoric, Lamya Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.
- [22] Google. 2020. *Bazel.* <https://bazel.build/>.
- [23] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *International Symposium on Software Testing and Analysis*. 223–233.
- [24] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. 2002. *The Vesta Software Configuration Management System*. Research Report. <http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-177.pdf>.
- [25] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Automated Software Engineering*. 426–437.
- [26] Sam Kamin, Lars Clausen, and Ava Jarvis. 2003. Jumbo: Run-time Code Generation for Java and Its Applications. In *International Symposium on Code Generation and Optimization*. 48–56.
- [27] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594.
- [28] Qingzhou Luo, Farah Hariri, Lamya Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653.
- [29] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. 2012. The Evolution of Java Build Systems. *Empirical Software Engineering* 17, 4–5 (2012), 578–608.
- [30] Atif M. Memon and Myra B. Cohen. 2013. Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness. In *International Conference on Software Engineering*. 1479–1480.
- [31] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proc. ACM Program. Lang.* 2, International Conference on Functional Programming (2018).
- [32] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *International Symposium on Foundations of Software Engineering*. 496–499.
- [33] Vladimir Nikolov, Rüdiger Kapitza, and Franz J Hauck. 2009. Recoverable Class Loaders for a Fast Restart of Java Applications. *Mobile Networks and Applications* 14, 1 (2009), 53–64.
- [34] Voas JM, Offutt J, Pan J. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *International Conference on Testing Computer Software*. 111–123.
- [35] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *Transactions on Software Engineering* 22, 8 (1996), 529–551.
- [36] Peter Smith. 2011. *Software Build Systems: Principles and Experience*. Addison-Wesley Professional.
- [37] Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming*. Springer Berlin Heidelberg. 30–50.
- [38] tevemadar. 2018. *Blocking on stdin makes Java process take 350ms more to exit.* <https://stackoverflow.com/a/48979347>.
- [39] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *International Symposium on Foundations of Software Engineering*. 805–816.
- [40] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Conference on Programming Language Design and Implementation*. 160–173.
- [41] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.
- [42] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *International Symposium on Software Testing and Analysis*. 385–396.

# Feedback-Driven Side-Channel Analysis for Networked Applications\*

İsmet Burak Kadron

University of California Santa Barbara  
Santa Barbara, CA, USA  
kadron@cs.ucsb.edu

Nicolás Rosner

University of California Santa Barbara  
Santa Barbara, CA, USA  
nrosner@gmail.com

Tevfik Bultan

University of California Santa Barbara  
Santa Barbara, CA, USA  
bultan@cs.ucsb.edu

## ABSTRACT

Information leakage in software systems is a problem of growing importance. Networked applications can leak sensitive information even when they use encryption. For example, some characteristics of network packets, such as their size, timing and direction, are visible even for encrypted traffic. Patterns in these characteristics can be leveraged as side channels to extract information about secret values accessed by the application. In this paper, we present a new tool called AutoFeed for detecting and quantifying information leakage due to side channels in networked software applications. AutoFeed profiles the target system and automatically explores the input space, explores the space of output features that may leak information, quantifies the information leakage, and identifies the top-leaking features.

Given a set of input mutators and a small number of initial inputs provided by the user, AutoFeed iteratively mutates inputs and periodically updates its leakage estimations to identify the features that leak the greatest amount of information about the secret of interest. AutoFeed uses a feedback loop for incremental profiling, and a stopping criterion that terminates the analysis when the leakage estimation for the top-leaking features converges. AutoFeed also automatically assigns weights to mutators in order to focus the search of the input space on exploring dimensions that are relevant to the leakage quantification. Our experimental evaluation on the benchmarks shows that AutoFeed is effective in detecting and quantifying information leaks in networked applications.

## CCS CONCEPTS

- Security and privacy → Software and application security; Web application security.

## KEYWORDS

Side-channel analysis, dynamic program analysis, network traffic analysis, input generation

### ACM Reference Format:

İsmet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-Driven Side-Channel Analysis for Networked Applications. In *Proceedings*

\*This material is based on research supported by NSF under Grants CCF-1901098 and CCF-1817242.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397365>

of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397365>

## 1 INTRODUCTION

As sensitive information migrates to online services, information leaks are becoming an urgent threat, and *side-channel* leaks, where private information can be extracted by analyzing visible side effects of computation, are becoming increasingly important. Well-known side-channel attacks include those based on power consumption [28], electromagnetic radiation [20], cache timing [48], and CPU-level branch prediction and race conditions, such as the Spectre [26] and Meltdown [30] attacks.

Although information leaks due to design flaws in hardware have been studied more extensively [3, 25, 31], *software-based* side-channels have also been reported [13, 27, 45, 47]. To prevent disastrous software-based side-channel attacks, we need tools that can automatically analyze software systems, detect software side channels, and assess the severity of the information leakage.

Recent work on software side channel detection [4, 8, 11, 32] focuses on white-box techniques. However, the source code of the system may not always be available. Even when it is, many systems do not lend themselves well to white-box analysis. Modern software systems often comprise multiple components joined by networks: clients, servers, peers. They may also include distributed storage, load balancing, microservices, etc. Components are often written in different programming languages, whereas most white-box tools target only one language. White-box tools often require significant manual effort to extract a slice of the system amenable to analysis and mock the rest. Such slicing is costly, error-prone, and may hide side channels. For example, a system may contain a timing vulnerability due to differences in the response time of one component. The side channel goes undetected when that component is stubbed out. This makes a strong case for black-box side-channel analysis tools that execute the whole system without modifications.

We present AutoFeed, a tool for feedback-driven black-box profiling of software systems that detects and quantifies side-channel leakage automatically. The user provides some seed inputs for the target system, and a set of mutators which, given a valid input, return another one. The user chooses a *secret of interest*—some aspect of the input that they consider sensitive, whose leakage they want to detect and quantify. AutoFeed then repeatedly executes the target system, generates new inputs, captures network traffic, and adjusts input generation and system execution strategies based on the feedback it obtains by analyzing captured traffic.

Modern systems use encryption. AutoFeed analyzes side channels in network traffic—the visible aspects of traffic that eavesdroppers can easily capture despite encryption, such as the size, timing, and direction of network packets. AutoFeed extracts meaningful features from these visible characteristics, and uses conditional entropy to find features that maximize information gain about the secret of interest. For example, it may find that the time elapsed between certain packets leaks some amount of information about the secret. The final output from AutoFeed is an automatically generated *ranking* of the top  $n$  most-leaking features, sorted by how much information they each leak about the secret of interest.

There has been prior work on quantifying leakage in network traces in particular [40, 47] and in program traces in general [14, 15]. However, all of them rely on manually generated input suites and do not address the problem of the quality of the input suite. AutoFeed automates the manual effort of providing inputs. Instead, the user writes mutators to explore the input space. An automated feedback loop progressively generates and runs more inputs and improves accuracy of leakage estimation. AutoFeed also automates the assessment of usefulness of different mutators and the stop criterion that determines when the leakage estimation and the output feature ranking become stable, avoiding diminishing returns of computational effort. Compared to prior work, AutoFeed enhances the degree of automation significantly, reduces the amount of wasted profiling effort, and improves the reliability of the results. Our contribution in this paper is to present a feedback-driven, black-box technique to detect and quantify side channels using mutator-based input generation, statistical modeling of the observed data, and a stop criterion to detect convergence of leakage estimation. AutoFeed runs incrementally, generates more inputs as needed, caches inputs to avoid repetitions, and stops running when its iterative leakage quantification stabilizes. In particular we present:

- (1) An automated search mechanism to determine crucial hyper-parameter values dynamically based on feedback, in order to estimate probability distributions for modeling the observed data, and a comparative study of techniques to model the observed data using histograms, Gaussian distributions, and kernel density estimation (KDE).
- (2) A mechanism to focus input space exploration on dimensions that provide more information about the leakage. AutoFeed lets users model the input space using mutators, and relate them to different dimensions of the input space. AutoFeed automatically explores each dimension and assigns weights to the mutators. The effort invested in exploring each dimension is proportional to how much each dimension fosters changes in leakage estimation.
- (3) An automated stop criterion that halts the input generation process once the leakage estimate stabilizes. This allows convergence of the leakage estimation to a value close to the ground truth independent of the starting input set.
- (4) Experimental evaluation of the effectiveness of AutoFeed on handcrafted examples with known quantitative ground truths and on the DARPA Space/Time Analysis for Cybersecurity (STAC) benchmark [17], which consists of realistic-sized software systems (Web, client-server, and peer-to-peer) developed by DARPA, in both controlled, low latency and less controlled,

high latency network conditions in order to evaluate side-channel vulnerability detection techniques.

The rest of the paper is organized as follows. In Section 2 we provide motivation and an overview of our approach. In Section 3 we describe the core techniques and heuristics we developed for AutoFeed. In Section 4 we describe our implementation. In Section 5 we present an experimental evaluation of AutoFeed. In Section 6 we discuss the related work. In Section 7 we conclude the paper.

## 2 MOTIVATION AND OVERVIEW

Generating a set of profiling inputs to quantify information leakage presents unique challenges. The problem is quite different from generating an input suite for testing. In traditional testing, the goal is to find inputs that violate assertions or crash the system. In side-channel profiling, the goal is to characterize the relationship between a certain *secret* (i.e., some private or sensitive variable) and the *publicly observable output* of the system, such as the timing and sizes of encrypted network packets. Many new issues arise. We do not know how inputs and outputs are related. We do not know how outputs and secrets are related. Each observable feature may reveal very little or very much about a secret. For each secret, there is an immense space of output features that could leak information about it—the timing of a particular network packet, the time elapsed between two packets, the size of a packet, the sum of sizes of a subset of the packets, etc. Given an observable output feature, it is hard to figure out how its value relates to the value of the secret.

**Challenge: Foster collisions.** Suppose a secret is picked. Given a set of inputs, each with a different value of the secret, suppose we run each input through the target system. If the set is small, we will find some feature (say, the time of a certain network packet) that takes a unique value for each secret value. Based on such observations, one might be misled into concluding that the feature fully leaks the value of the secret. But the actual leakage could be much lower, or even none, because: (1) If we generate more inputs, we may observe the same value of the feature for two inputs with different secrets. We call these *collisions*. (2) If we run the same input twice, due to system *noise*, we may see different feature values for the exact same input. These two phenomena, collisions and noise, create complex relationships between secrets and features.

It is desirable to find inputs that foster collisions between secrets in each of the system's observable output features. Imagine that we probe a medical system to see how much information it leaks about a patient's age when a patient's record is accessed by medical staff through the network. If we profile the system with a small sample (e.g., fetch 10 patient records), we may observe that the size of a certain packet changes with the age of the patient. But the size of the packet could have taken a unique value for each of the 10 executions by coincidence. A collision occurs when we fetch the records of two patients with different ages (say, 18 and 57) for which that packet has the same size (say, 215 bytes). This introduces uncertainty: an eavesdropper that captures an interaction with a 215-byte packet cannot tell if the patient is 18 or 57 years old. Thus, the feature does not *fully* leak the secret. As we fetch more records, if the observed collision rate progressively approaches the actual rate, our quantification of information leakage will progressively approach the actual amount of information leaked. An input set

with an overly low collision rate (w.r.t. the full input space) will result in overestimating the leakage. Finding inputs that foster collisions in the most-leaking features improves the estimation.

Now consider time features, such as the time elapsed between the third and fourth packets of each interaction. We want to quantify how much information this feature leaks about patient age. Since time is continuous, the probability of seeing the exact same value for two inputs is zero. Even if we run *the exact same input* multiple times, we will see slightly different values. This is due to system noise, such as variance in network latency. By running each input multiple times, we can model the noise as a probability distribution. Collisions occur when distributions overlap. The greater the overlap, the more uncertainty about the secret value, resulting in a lower estimation of the amount of information leaked.

Note that the relationships between inputs, outputs and secrets are arbitrary: they depend on the behavior of the target system. The same is true of noise, and software pseudo-randomness can add arbitrary extra noise. Hence, there are no general rules to build an adequate black-box input suite before the analysis. Statically crafted input suites can always lead to incorrect results.

**Challenge: Explore a vast input space.** To compute the exact amount of information leaked by a system about a secret when performing an action, we would need to execute the action for every input, which is generally not feasible. How many inputs we are willing to execute depends on how long it takes to run each one and how long we are willing to wait for the analysis to complete.

System inputs can be complex and may include structured data. For example, the AIRPLAN system from the DARPA STAC benchmark (see Section 5.2) takes as input an arbitrary graph of airports and flight routes, and each edge is decorated with six different weights. Inputs to DARPA’s RAILYARD system involve different kinds of train cars, different types and quantities of cargo, crew members, train routes, stops, schedules, and more. The possibilities are endless and depend on the system. Each output feature can be affected by *any part* of the input—including those that are related to the secret of interest, and those that are not.

Prior work [14, 15, 40, 47] requires the user to provide the full input suite *before the analysis begins*. Thus, the user must sample the input space in some way that covers all its dimensions adequately. But, even for one feature, the user cannot know in advance which input dimensions will foster changes in the leakage estimation of that feature. To make things worse, there is an enormous space of observable features. If the user tries to be conservative and cover all bases, combinatorial explosion results in a prohibitive number of inputs. If the user tries to reduce the input set to keep the analysis time feasible, leakage estimation results may be incorrect.

**Challenge: Quantify the leakage.** Extracting probability distributions from observable features is nontrivial. Histograms can overfit the data and lead to false positives. Gaussian fitting can overabstract the data: if it is not normally distributed, the model will be wrong. For example, when a feature is multi-modal, Gaussian fitting will produce a unimodal approximation, and false overlaps will underestimate the leakage. Kernel density estimation [34] offers greater flexibility and is well-suited for a wide variety of data, but heavily depends on the *window size* or bandwidth; too small a value leads to similar problems as with histograms, whereas too large a value can lead to similar problems as with Gaussian fitting.

**Overview of our approach.** As said above, crafting an input suite before the analysis is tedious and risky. Different input suites can lead to different leakage quantification results. AutoFeed offers a mutation-based mechanism (see 4) to specify the space of valid inputs. It automatically generates new inputs on demand using a feedback loop. Manual user effort is limited to writing the mutators, providing a small set of initial seeds, and choosing the secret of interest. (The mutators, once written, can be reused for many secrets.) AutoFeed automates everything else. It iteratively mutates inputs and periodically quantifies the leakage to update its belief about which features leak the greatest amount of information about that secret. In doing so, it automates both the exploration of the output feature space and the exploration of the input space. Since the user cannot know which mutators will be most important for a secret, AutoFeed measures the effect of different mutators on leakage estimation, and weighs them accordingly (see 3.3) in a feedback-driven way. By focusing the computational effort on those mutators that have greater effect on the leakage estimation of the most promising features, the input space is explored efficiently.

Quantification computation is nontrivial. We conducted a comparative analysis of different approaches (see 5.4). AutoFeed automatically discovers the distribution of observed data using KDE and automatically finds a suitable bandwidth parameter (see 3.4).

By enforcing a stop criterion, AutoFeed automates evaluating whether the leakage estimation is stable enough (see 3.5). This reduces the risks associated with having to manually decide when to stop. It also allows AutoFeed to run analyses batches unattended.

### 3 FEEDBACK-DRIVEN SIDE-CHANNEL ANALYSIS

In this section we provide some basic definitions and explain the main algorithms and heuristics used in AutoFeed.

#### 3.1 System Model

Assume that a software system, use case, and secret of interest are selected by the user. We reuse the following definitions from the system model in [40]. The *input domain*  $\mathbb{I}$  is the set of all valid inputs for the use case. The *secret domain*  $\mathbb{S}$  is the set of all values that the secret of interest can take. Given an input, the *secret function*  $\zeta : \mathbb{I} \rightarrow \mathbb{S}$  projects its secret value. Running every input in  $\mathbb{I}$  is usually not feasible: the *input set*  $\mathbf{I} \subseteq \mathbb{I}$  is the set of distinct inputs that are executed during an analysis. Since the secret is a function of the input, by choosing a set of inputs, we are also choosing a set of secrets. The *secret set*  $\mathbf{S} \subseteq \mathbb{S}$  is the set of distinct secrets that appear in some input during an AutoFeed analysis. Assuming a generalized  $\zeta : \mathcal{P}(\mathbb{I}) \rightarrow \mathcal{P}(\mathbb{S})$ , we can say that  $\zeta(\mathbf{I}) = \mathbf{S}$ . A *packet* is an abstraction of a real network packet. We assume packets are encrypted. Decrypting them is beyond the scope of this work. We consider side-channel characteristics of each packet: its size, time, and direction in which it flows. Each time we execute an input  $i \in \mathbb{I}$  through the system, we capture a *network trace*, which is a sequence of packets. We also add the following definitions. A *seed* is an input  $i \in \mathbb{I}$  provided by the user. A *mutator* is a function  $m : \mathbb{I} \rightarrow \mathbb{I} \cup \{\text{None}\}$  that, given a valid input, returns another valid input, or None if the input cannot be mutated by  $m$ . For instance, in our AIRPLAN example, the RemoveFlight mutator removes one

of the direct flights between two airports. This mutator cannot be applied to a map in which all direct flights have been removed. Lastly, an *initial set* of inputs  $D \subseteq \mathbb{I}$  is a set of inputs obtained by applying some amount of random mutation to the seeds.

**Procedure 1** AUTOFEED( $App, I, M, RPI, C$ ) Given an application  $App$ , an initial set of inputs  $I$ , a set of mutators  $M$ , a repetition per input value  $RPI$ , and a time budget per iteration  $C$ , AUTOFEED quantifies the leakage using a feedback loop.

```

1:  $Traces \leftarrow EXECUTE(App, I, RPI)$                                 ▷ Generate corresponding traces
2:  $N \leftarrow C / (\text{AvgTime}(Traces) \times RPI)$                          ▷ Calculate the number of inputs ( $N$ ) to generate per iteration, given  $C$  seconds of time budget per iteration
3:  $I' \leftarrow MUTATE(I, M, N, \vec{W}_{uniform})$  ▷ Generate new inputs using mutators where each partition of mutators has equal weight
4:  $Traces' \leftarrow EXECUTE(App, I', RPI)$                                 ▷ Generate corresponding traces
5:  $I \leftarrow I \cup I'$ 
6:  $Traces \leftarrow Traces \cup Traces'$ 
7:  $Leak' \leftarrow QUANTIFYLEAKAGE(Traces)$ 
8:  $\langle \vec{W} \rangle \leftarrow GETWEIGHTS(App, I, M, N)$  ▷ Compute the weights for the mutators
9: repeat                                                               ▷ Main loop for feedback-driven exploration
10:    $Leak \leftarrow Leak'$ 
11:    $I' \leftarrow MUTATE(I, M, N, \vec{W})$                                 ▷ Generate new inputs using mutators
12:    $Traces' \leftarrow EXECUTE(App, I', RPI)$                                 ▷ Generate corresponding traces
13:    $I \leftarrow I \cup I'$ 
14:    $Traces \leftarrow Traces \cup Traces'$ 
15:    $Leak' \leftarrow QUANTIFYLEAKAGE(Traces)$ 
16: until  $|Leak' - Leak| < \epsilon$  ▷ Stop criterion check convergence of leakage value
17: return  $Leak'$ 
```

**Procedure 2** GETWEIGHTS( $App, I, M, N$ ) Given an application  $App$ , a set of inputs  $I$ , a set of mutators  $M$  and a partition, and number of inputs to generate  $N$ , GETWEIGHTS computes weights for subsets of mutators.

```

1:  $Traces \leftarrow EXECUTE(App, I, RPI)$                                 ▷ Generate corresponding traces
2:  $F \leftarrow EXTRACTFEATURES(Traces)$  ▷ Extract features over traces of original inputs
3: for each subset  $M_i$  of  $M$  do                                         ▷ where the  $M_i$  are a partition of  $M$ 
4:    $I_i \leftarrow MUTATE(I, M_i, N)$                                 ▷ Generate inputs using a subset of mutators
5:    $Traces' \leftarrow EXECUTE(App, I_i, RPI)$ 
6:    $F' \leftarrow EXTRACTFEATURES(Traces')$  ▷ Extract features over traces of mutated inputs to estimate weight of  $M_i$ 
7:    $\vec{W}[i] \leftarrow \sum_j |F'_j - F_j| / (F_{max} - F_{min}) + [Sec'_j \neq Sec_j]$  ▷ Weight of the current subset of mutators is proportional to number of mutated inputs with a different feature value or secret
8:    $W_{sum} \leftarrow \sum_i W[i]$ 
9: for each  $\vec{W}[i]$  do                                                 ▷ Normalize the mutator weights
10:    $\vec{W}[i] \leftarrow \vec{W}[i] / W_{sum}$ 
11: return  $(\vec{W})$ 
```

**Procedure 3** MUTATE( $I, M, N, \vec{W}$ ) Given a set of inputs  $I$ , a set of mutators  $M$ , number of inputs to generate  $N$ , and mutator weights  $\vec{W}$ , MUTATE generates new unique inputs using the mutators.

```

1:  $I_{new} \leftarrow \emptyset$ 
2: while  $|I_{new}| < N \wedge |I| > 0$  do
3:    $i \leftarrow RANDOMSELECT(I)$                                          ▷ Select a random input
4:    $M' \leftarrow M$ 
5:    $done \leftarrow false$ 
6:   while  $M' \neq \emptyset \wedge \neg done$  do
7:      $m \leftarrow RANDOMSELECT(M', \vec{W})$  ▷ From set  $M'$  according to weights  $\vec{W}$ 
8:      $i_{new} \leftarrow m(i)$ 
9:     if  $i_{new} \in I \vee i_{new} = None$  then
10:        $M' \leftarrow M' - \{m\}$  ▷ If a mutator does not create a new input, drop it
11:     else
12:        $I_{new} \leftarrow I_{new} \cup \{i_{new}\}$ 
13:        $done \leftarrow true$ 
14:     if  $\neg done$  then
15:        $I \leftarrow I - \{i\}$                                               ▷ If no mutator yields a new input, drop it
16: return  $I_{new}$ 
```

## 3.2 AutoFeed Workflow

The high level algorithm demonstrating the workflow of the AutoFeed tool is shown in Procedure 1. AutoFeed requires the following inputs from the user: an application to run  $App$ , initial seed inputs  $I$ , a set of mutators  $M$ , value for repetitions per input  $RPI$ , and a time budget per iteration  $C$ . First, AutoFeed executes the  $App$  with the initial seed inputs to generate an initial set of traces. Based on these initial traces, it calculates the number of inputs to generate per iteration ( $N$ ) that corresponds to the given input time budget per iteration ( $C$ ). Then, it applies the mutators on the seed inputs to get new inputs, executes the  $App$  on these inputs, and uses the traces obtained from these executions to obtain an initial estimation of the information leakage. Using the initial leakage results, AutoFeed uses heuristics to compute weights for mutators, where the weight of each mutator corresponds to the likelihood of applying that mutator during input generation. After these initialization steps, AutoFeed starts executing its main loop for feedback-driven exploration of the input state space for obtaining an accurate estimation of information leakage. In each loop iteration, AutoFeed uses mutators to generate new inputs, executes the  $App$  on new inputs to generate corresponding traces, and updates the leakage estimation using all the traces captured so far. When the change in the leakage estimate falls below a small value ( $\epsilon$ ), AutoFeed terminates execution and reports the computed leakage.

In the main workflow of the AutoFeed tool shown in Procedure 1 we use two other procedures that we discuss below: GETWEIGHTS, and MUTATE. For the sake of readability and clarity of presentation, we present all these procedures from the perspective of a single feature (the top feature) corresponding to the feature that leaks the most amount of information. In actual implementation of AutoFeed, a large set of features are taken into account and their leakage is estimated until termination. After the initial input generation step and initial leakage estimation, only for GETWEIGHTS top  $k$  features are selected as we believe mutators that discover more behaviors on those features will impact the leakage results.

## 3.3 Assigning Weights to Subsets of Mutators

AutoFeed uses the user-provided mutators to generate new inputs and explore the input space. It is not possible to know in advance which mutators would be more effective in exploration of the information leakage. Some mutators may generate new secret values which may help our analysis by improving the information leakage estimation. Some mutators may generate inputs with the same secret value but different feature values which can again help our analysis by improving the information leakage estimation. On the other hand, some mutators may generate inputs that do not provide any new insight to the relationship between the secret and the observable features. For example, some mutators may change the input without modifying the secret or any of the observable features. Such mutators will not help our analysis in improving the information leakage estimation. AutoFeed evaluates the influence of mutators on the leakage estimation based on changes in top feature or secret and computes weights for mutators which are proportional to their likelihood of changing secret value or perturbing feature values. These weights are then used to bias the random selection of the mutators where each mutator is selected with a

probability that is proportional to its weight. Hence, the mutators that influence the leakage estimation less are chosen less frequently and the mutators that influence the leakage estimation significantly are chosen more frequently.

To do this analysis, the user groups the mutators into subsets. We call these subsets of mutators *dimensions*. Mutators can be grouped by the attribute they are modifying. For instance, in the RAILYARD system, mutators that add/remove stops from the train schedule are one dimension, whereas those that add/remove personnel from the train crew are another dimension. Mutators can also be grouped by the magnitude of the change that they cause on the input: if a mutator increases an input field by 1, and another mutator increases it by 1000, we may want them to be weighted separately.

We assume that the user provides a partition of the set of mutators, so that each mutator belongs to a single dimension, and each dimension is a subset of the set of mutators. To assess the impact of each subset of mutators on the leakage estimation, for each subset, we generate and run inputs generated only using mutators in that particular subset. Using the traces of these runs and previous traces, we quantify the leakage and record the amount of change in the leakage between this step and the previous step. After we do this test for each subset of mutators, we weigh each subset proportionally to the amount of change in leakage we recorded for that subset of mutators. Pseudocode for this process is given in Procedure 2.

### 3.4 Leakage Quantification

This section describes how QUANTIFYLEAKAGE function in Procedure 1 works. To quantify information leakage, we start from a set of captured traces, each one labeled with the secret value associated with that trace, and we align packets using markers inserted at runtime which denote different stages of the interaction. We then extract the related packet based features (such as packet timing and size) and aggregated features (such as total duration, total size, etc.) obtained using alignment. After obtaining the features, we can estimate the probability distribution of features per secret using multiple methods and compute the mutual information between the secret and feature using the estimated probability distribution for each feature. We use *Shannon entropy* [43] to calculate the mutual information  $I(S; V)$  and it is derived as

$$I(S; V) = - \sum_{s \in S} p(s) \log_2 p(s) - \left( - \sum_{v \in V} p(v) \sum_{s \in S} p(s|v) \log_2 p(s|v) \right)$$

where  $S$  and  $V$  are the sets of secret and feature values and we estimate  $p(v|s)$  for each secret,  $p(s)$  is assumed to be uniform and  $p(s|v)$  and  $p(v)$  are estimated using Bayes' rule. The first term represents the initial amount of information about the secret. The second represents the remaining uncertainty after observing the feature. The difference is the amount of information gained by observing that feature. For more details, see [40, 43].

The simplest way of estimating the shape of the data distribution is by modeling it as a histogram. This method puts the data in discrete bins where the ratio of elements determine the probability. One problem with this method is that its results are dependent on the bin size and determining an ideal bin size is difficult. If we conservatively choose the smallest bin size we can, then collisions will go undetected unless a huge number of samples is used.

Another method is modeling the data distribution as a Gaussian distribution where the mean  $\hat{\mu}$  and standard deviation  $\hat{\sigma}$  of the data is obtained and  $\hat{p}(x)$  is estimated as  $N(x; \hat{\mu}, \hat{\sigma})$ . This method extrapolates well but is based on the strong assumption that the data is normally distributed. This assumption may fail if the data is generated from a more complex distribution. Whenever the assumption fails, the data is underfitted: spurious collisions arise, and the information leakage tends to be understated.

Another way of estimating probability distributions is using *kernel density estimation* (KDE) [34]. Using KDE, we can estimate the distribution of data without assuming a specific distribution. Unlike a histogram, our estimation is smooth, which helps us model continuous data better and extrapolate to unseen data more easily. If we want to estimate  $p(x)$ , the kernel density estimator  $\hat{p}(x)$  is

$$\hat{p}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

where  $n$  is number of samples,  $h$  is the positive bandwidth parameter and  $K$  is a non-negative function called *kernel*. There are various kernel functions: uniform, triangular, Gaussian, Epanechnikov, etc. The bandwidth  $h$  affects our estimation greatly. If it is too small, it overfits the data we have; if it is too large, it underfits the data.

In this work, we have used two methods for bandwidth selection. First selection method is the optimal bandwidth if the underlying distribution is Gaussian in which bandwidth  $h = 1.06\hat{\sigma}n^{-1/5}$ , where  $\hat{\sigma}$  is the standard deviation of the data [44]. Second method is more general and instead of assuming any underlying distribution, we use statistical cross-validation techniques to select the ideal bandwidth. We use grid search which is used for hyper-parameter optimization by training using a set of candidate parameters on a model (KDE in this case) and evaluating each trained model. The evaluation metric is obtained using *repeated k-fold cross-validation* where the data is split into  $k$  equal subsets and for a single subset, we use the other  $k - 1$  subsets to train the model by estimating KDE using only the other subsets. The selected subset is tested to obtain the likelihood of this subset on the model. If the likelihood is high, that means KDE with this particular bandwidth does not overfit the data points and generalizes to unseen data as we test likelihood with a separate subset from training subsets. This process is repeated for all  $k$  subsets, for multiple splits of the data and the results (likelihood) are averaged. The model which gets us a higher likelihood on the test sets is the best performing model as it means it fits the data well. We select the ideal bandwidth as the bandwidth of the best performing model [7, 21, 41]. This method has some variance in bandwidth selection as it depends on the dataset and the particular splitting but variance can be reduced with the repetitions. [9] We set  $k$  to be 5 in our experiments, with 3 repetitions.

For comparison purposes, we have also included in the experiments a version of KDE in which the bandwidth is fixed. As for kernel selection, we use Epanechnikov kernel in our implementation which is optimal in minimizing mean square error. [50]

### 3.5 Stop Criterion

As AutoFeed's main loop runs, the leakage estimation can converge if newly generated inputs no longer discover new behaviors. If the estimation stops changing, this can mean that the exploration of the

input space has saturated and we may finish the analysis and print the leakage estimation ranking. To detect this condition, we check the change in information leakage of the top leaking feature and finish the analysis if it is smaller than a predetermined  $\epsilon$  for a long enough period of time. Accuracy of leakage estimation depends on the accuracy of probability estimation  $\hat{p}(x)$  and as number of inputs  $N$  increases, accuracy of  $\hat{p}(x)$  will also increase.

In Procedure 1 we show the pseudocode for this process. Note that this pseudocode is a simplified version: in the actual AutoFeed implementation, we terminate the analysis only if leakage estimation for the top  $k$  features converges, and we assume that leakage estimation converges if it changes less than  $\epsilon$  for at least  $n$  consecutive iterations (rather than the last iteration as in Procedure 1), where  $k$  and  $n$  are adjustable parameters. To simplify the presentation, in Procedure 1 we show a version where  $k = 1$  and  $n = 2$ , but the values of  $k$  and  $n$  are adjustable in our implementation.

#### Listing 1: Example usage of AutoFeed API

```
from autofeed import Platform, Container, Sniffer, App, Input, Mutator

class ExampleApp(App):
    def launch(self):
        Platform.cleanuphosts(["homer.example.edu", "marge.example.edu"])
        # Deploy two containers on two different machines
        self.servercontainer = Container("example/server:v1.0")
        self.clientcontainer = Container("example/client:v1.0")
        self.server = Platform.launch(self.servercontainer, "homer.example.edu")
        self.client = Platform.launch(self.clientcontainer, "marge.example.edu")
        # Run the server
        server_cmd = "bash -c 'cd /home/server && ./startServer.sh'"
        self.server.exec(server_cmd, detach=True)
    def shutdown(self):
        self.server.killrm()
        self.client.killrm()
    def run(self, inputs):
        sniffer = Sniffer(ports=[8080, 8081])
        sniffer.start()
        for input in inputs:
            sniffer.startinteraction(input.secret())
            self.client.createfile(input, "/home/client/input.txt")
            cmdfmt = "bash -c 'cd /home/client && ./startClient.sh {} {}'"
            self.client.exec(cmdfmt.format("homer.example.edu", "input.txt"))
        sniffer.stop()
        return sniffer.traces()

class ExampleAppInput(Input):
    def __init__(self, num_people, temperature):
        assert num_people >= 0
        self.num_people = num_people
        self.temperature = temperature
    def __eq__(self, other):
        return self.num_people == other.num_people
        and self.temperature == other.temperature
    def __hash__(self):
        return hash((self.num_people, self.temperature))

class IncreaseNumberOfPeople(Mutator):
    def mutate(self, input):
        input.num_people += 1
        return input

class DecreaseNumberOfPeople(Mutator):
    def mutate(self, input):
        if input.num_people > 0:
            input.num_people -= 1
            return input
        else: # we do not allow a negative number of people
            return None

class IncreaseTemperature(Mutator):
    # ... etc ...
```

## 4 IMPLEMENTATION

AutoFeed is written in Python. We use the trace-capturing library from [40], which relies on *scapy* [6] for packet sniffing. AutoFeed uses *scikit-learn* [36] and *numpy* [2] for probability estimation and

leakage quantification, *matplotlib* [23] for plotting, and the Python *docker* [1] library for container orchestration. We ran AutoFeed on the DARPA STAC Reference Platform, which comprises three Intel NUC computers (see Section 5.3). Users can run AutoFeed on any number of computers and networks, including localhost. AutoFeed provides Python base classes, templates, and examples for:

**Orchestration.** We use Docker [18] for deployment and launching of components (clients, servers, peers) on different machines. We provide a Container abstraction to launch container  $C$  on host  $H$ , copy a file  $F$  to  $C$ , run a command  $K$  on  $C$ , and shut down  $C$ .

**System setup and execution.** The App class is a base class that represents a black-box system. Users can subclass App to add their own systems. An App must provide three things: a launch method that launches the system; a shutdown method that shuts it down; and a run method to execute inputs. Given a list of Input instances, the App.run(inputs) method should return a list of traces.

**Packet sniffing.** AutoFeed provides a Sniffer object that offers a simple interface for capturing traffic and labeling the captured traces. Before starting each interaction, the App's run method should call Sniffer.startinteraction(secret) to ensure that the captured traffic is labeled with the correct secret.

**Input model.** The Input base class represents a valid input for an App. Users subclass Input and add members to model relevant characteristics of an input instance. The only two mandatory methods are eq (equality comparator) and hash. These are used by AutoFeed to hash previously seen input instances and avoid unwanted repetitions. When in doubt, a simple way to implement a reasonable hash method is to pack all relevant class members in a tuple and call Python's primitive hash method on that tuple. This ensures that any change in any member affects the hash code.

**Mutators.** The Mutator base class represents a mutator that transforms valid inputs. The only required method is mutate, a static method that takes an Input and returns another. The method assumes that the Input is valid and must return another valid one. If the mutation cannot be applied or makes no sense for that Input, the method should return None.

Listing 1 shows an example that uses these classes. For space reasons, the Input subclass only has two members. The AutoFeed codebase contains examples with varying degrees of complexity, including apps, inputs, and mutators for the STAC benchmark.

## 5 EXPERIMENTAL EVALUATION

We first experimentally evaluate AutoFeed using five example functions which have interesting input/output relationships. We also evaluate AutoFeed using software systems from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [16], which are publicly available [17]. The STAC systems are multi-component systems (Web, client-server, peer-to-peer) that communicate over TCP streams encrypted with TLS/SSL, developed by DARPA to evaluate side-channel vulnerability detection techniques. The applications in our benchmark are a superset of those used in [40]. We added RAILYARD because we were interested in modeling its highly structured input format with the mutator-based approach.

## 5.1 Example Functions

We define five example functions which take an input and produce a single feature in order to evaluate the contributions discussed in Section 3.3 and 3.4. Examples 1–4 have the input format  $(s, x, y, a, b, c, d, e, f, g, h)$  where  $s$ , the secret value, is between 1–16, and the other fields are between 1–100. Example 5 takes a list of strings as input and the secret value is the length of the list limited to a maximum length of 15. Since the secret has 16 possible values in all cases, the total amount of information that could possibly be leaked is  $\log_2 16 = 4.00$  bits. Code for Examples 1–5 can be seen in Listing 2. Feature value of Example 1 is distributed uniformly between 0.5 and 1.0. Since there is no correlation between secret and feature, this example leaks 0 bits. In Example 2, there’s a bijection between feature values and secrets. Thus, this example fully leaks 4.00 bits. Example 3 is multimodal, where the distribution changes according to value of  $x$ . When  $x$  is even, there is perfect correlation between secret and feature values. When  $x$  is odd, there is no correlation. We use Shannon entropy, an average measure of leakage, and this example leaks 2.00 out of 4.00 bits. Feature of Example 4 depends on fields  $x$  and  $y$  but those fields are not related to secret, thus this example leaks 0 bits. Feature of Example 5 depends on both number and length of list elements, thus there are some collisions. It leaks 2.21 bits of information.

**Listing 2: Code for the example functions**

```
def f1(s,x,y,a,b,c,d,e,f,g,h):
    return randomfloat(0.5,1.0)

def f2(s,x,y,a,b,c,d,e,f,g,h):
    return random(1,50)*20 + s

def f3(s,x,y,a,b,c,d,e,f,g,h):
    if x%2 == 0: return s*10
    else: return y+1000

def f4(s,x,y,a,b,c,d,e,f,g,h):
    return x+y

def f5(list1):
    return len(str(list1))
```

## 5.2 STAC Systems

AIRPLAN (265 classes, 1,483 methods) is the airline system from our Section 2 example. Users can upload, edit, and analyze flight routes by metrics like cost, flight time, passenger and crew capacities. Our secret of interest is the *number of airports* in a route map uploaded by a user. BIDPAL (251 classes, 2,960 methods) is a peer-to-peer system where peers buy and sell items via a single-round auction with secret bids. Users can create auctions, search auctions, and place bids. The secret of interest is the *secret bid* placed by a user. GABFEED (115 classes, 409 methods) is a Web-based forum. Users can create posts, search existing posts, and engage in chat. Our secret of interest is the *Hamming weight* (i.e., number of ones) of the server’s private key. SNAPBUDDY (338 classes, 2,561 methods) is a Web application for image sharing. Users can upload photos from different locations, share them with their friends, and find out who is online by geographical proximity. Our secret of interest is the *location* of a user (victim). POWERBROKER (315 classes, 3,445 methods) is a peer-to-peer system used by electricity companies to buy and sell power. Plants with excess power try to sell it, and plants

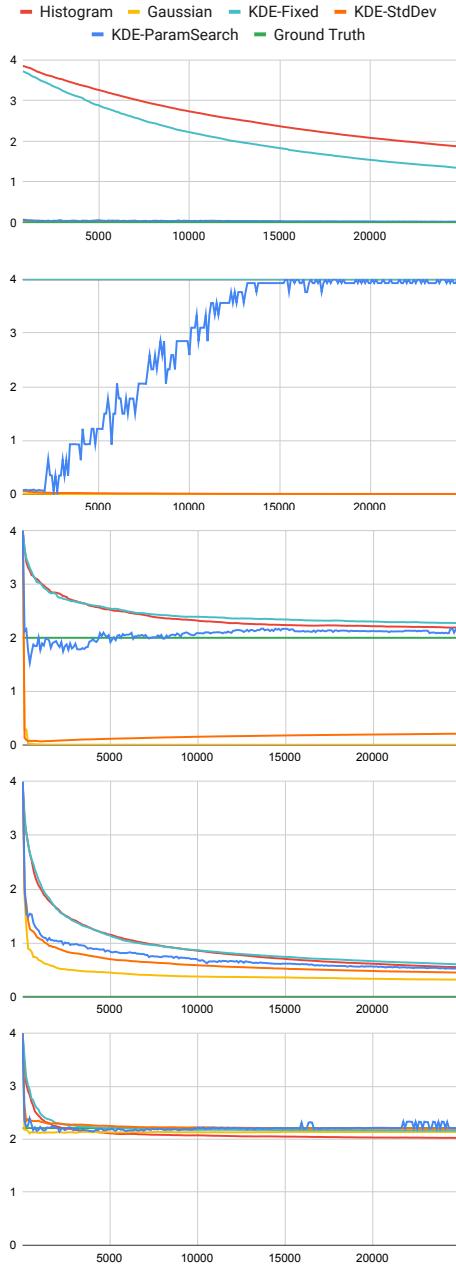
**Table 1: Mutators used (each line is a different dimension).**

<b>AIRPLAN</b>	
AddAirport, RemoveAirport	Add/remove one airport.
AddFlight, RemoveFlight	Add/remove one direct flight.
IncrDensity, DecrDensity	Increase/decrease flight density by 20%.
IncrWeight, DecrWeight	Increase/decrease one weight value by 1.
BoostWeights, DeboostWeights	Multiply/divide all weights by 10.
<b>RAILYARD</b>	
AddCar, RemoveCar	Add/remove a train car.
AddCargo, RemoveCargo	Add/remove a piece of cargo.
AddCrew, RemoveCrew	Add/remove one crew member.
AddStop, RemoveStop	Add/remove one train stop.
ChangeStops	Change all stops with new ones.
ChangeCrew	Change all crew with new ones.
<b>GABFEED</b>	
AddOne, RemoveOne	Add/remove one 1 to the key.
AddFive, RemoveFive	Add/remove five 1s to the key.
ShuffleOnes	Shuffle the 1s in the key.
<b>TOURPLANNER</b>	
ReplaceOneCity	Replace one city with a different one.
ShuffleCities	Shuffle the order of the five cities.
<b>BIDPAL</b>	
IncrBid, DecrBid	Increase/decrease bid by \$10.
<b>POWERBROKER</b>	
IncrOffer, DecrOffer	Increase/decrease the offer by \$10.
<b>SNAPBUDDY</b>	
PickLocation	Pick a known location from the list.

that need power try to buy it. The secret of interest is the *value offered* by one of the plants (victim). TOURPLANNER (321 classes, 2,742 methods) is a client-server tour optimizer—a variation of the traveling salesman problem. Given a list of cities that the user wants to visit, it computes a tour with optimal travel costs. The secret of interest is *the set of places* that the user (victim) wants to visit. RAILYARD (28 classes, 60 methods) is a system to manage a train station. The station manager can build trains by adding different kinds of cars, different types and quantities of cargo, adding personnel to the train, and adding stops to the train’s schedule. The secret of interest is *the set of types of cargo* that are on the train when it departs from the station.

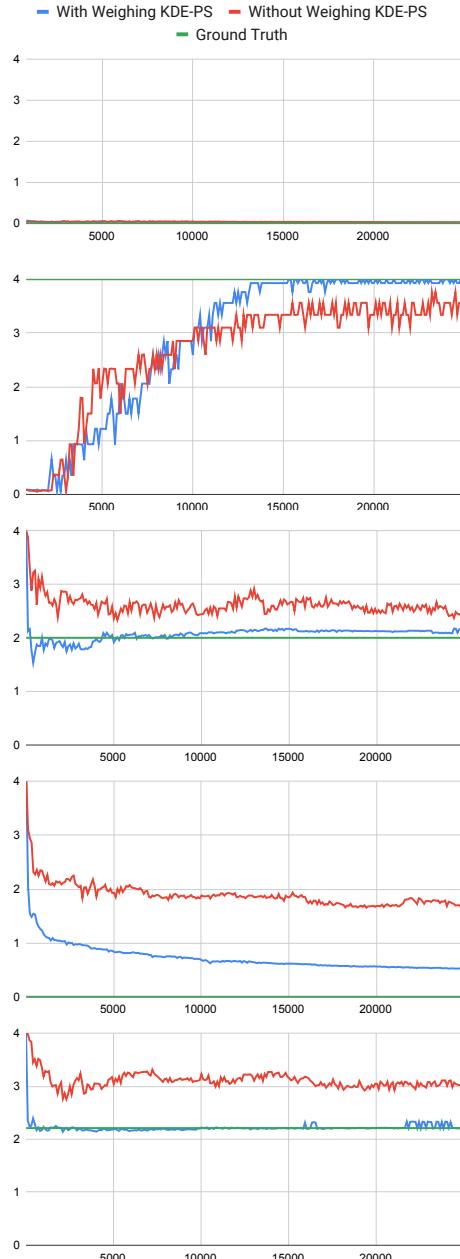
## 5.3 Experimental Setup

We used the DARPA STAC Reference Platform [40], with 3 Intel NUCs (server, client, and eavesdropper) connected by an Ethernet switch with low noise (latency: 0.22 ms min, 0.31 ms avg, 0.57 ms max). As for AutoFeed parameters, we set RPI to 20 for all programs when looking for timing side channels, and 5 for AIRPLAN 3 and SNAPBUDDY as there was non-determinism in their behavior. Time budget per iteration  $C$  is set to 5 minutes. We set the histogram bin size to 1 for space side channels, and  $10^{-5}$  for time. For KDE with fixed bandwidth, we set the bandwidth to 0.1 for space side channels, and  $10^{-5}$  for time. For grid search, we search over 10 parameters from the aforementioned fixed bandwidth for space/time to the maximum range of the relevant feature. We also include the standard deviation bandwidth in the search. For mutation weighing, we assign weights using GETWEIGHTS, considering top-5 features. The mutators for DARPA STAC systems are described in Table 1. The mutators are manually written to modify the secret and various



**Figure 1: Information leakage results for Examples 1–5 using Gaussian, Histogram and KDE. X-axis shows number of data points. Y-axis shows leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.**

aspects of the input with the hope that some of the mutators will affect the observables. The secret of interest for each app in DARPA STAC systems is determined by DARPA. For the stop criterion, we set the value of  $\epsilon$  to a 0.5% difference and checked that, for the top feature, the leakage estimation stayed within that difference for 3 consecutive iterations.



**Figure 2: Information leakage results for Examples 1–5 with and without mutator weighing, using KDE-ParamSearch. X-axis shows number of data points. Y-axis is leakage in bits. Ground truth for the Examples 1–5 are 0 bits, 4.00 bits, 2.00 bits, 0 bits and 2.21 bits respectively.**

We also used two leakage quantification tools, Leakiest [15] and F-BLEAU [14], for comparison. Leakiest computes mutual information between each feature and secret using histogram and KDE assuming Gaussian distribution for quantification and hypothesis testing. F-BLEAU computes min-entropy, which provides a lower bound on Shannon entropy, using a nearest neighbor based approach on all features.

## 5.4 Experimental Results

**Leakage method comparison.** For the five example functions, we started with 16 seed inputs and ran 250 iterations, obtaining 100 data points per iteration. Results are shown in Figure 1. In Example 1, where observables are continuous and uniform, Gaussian and KDE with std.dev. bandwidth converge easily. Histogram and KDE with fixed bandwidth converge very slowly. KDE with parameter search converges to the ground truth as fast as Gaussian and KDE-StdDev. In Example 2, Gaussian and KDE-StdDev wrongly converge to zero leakage: they assume a Gaussian distribution, but this feature is multi-modal. Histogram and KDE-Fixed converge to the correct result right away thanks to small bin size and bandwidth parameters. KDE-ParamSearch initially gets the wrong result but converges to the correct one when enough data is obtained. In Example 3, because the feature distribution is bimodal, Gaussian and KDE-StdDev yield incorrect results. Histogram and KDE-Fixed converge to a value near the actual leakage, but very slowly. KDE-ParamSearch converges much faster to the correct result, unlike the other methods. In Examples 4–5, all methods perform similarly.

In all five cases, when an assumption fails, the method yields a wrong result or takes too long. Using KDE-ParamSearch, our results do not overfit the data like Histogram or KDE-Fixed, and they do not underfit like Gaussian and KDE-StdDev. With this approach, we are able to select the best bandwidth value that maximizes likelihood of data and we are able to converge to the correct leakage value.

For STAC applications, using a small set of seeds (<75), we are able to distinguish if a vulnerability is present, mitigated, or absent; weigh mutators automatically, and stop iterating when the leakage values for top features stabilize. See Table 2.

For AIRPLAN, RAILYARD and SNAPBUDDY, AutoFeed converges quickly and vulnerable cases are found to leak 100%, whereas in cases where leakage is mitigated or absent, lower leakage results are found. For all cases except RAILYARD, the  $L_{KDE-PS}$  result is greater than the lower bound estimated by F-BLEAU. F-BLEAU estimates leakage for multi-dimensional feature vectors and it may have found a correlation between 2 features that AutoFeed is not able to detect since AutoFeed analyzes each feature separately.

For GABFEED, POWERBROKER, BIDPAL and TOURPLANNER, AutoFeed converges in 8 to 20 iterations and the leakage results for leaky versions have higher leakage than for non-leaky versions. For POWERBROKER, BIDPAL and GABFEED cases, especially in non-leaky cases, Histogram and KDE-Fixed overestimate the leakage. For POWERBROKER 1 and TOURPLANNER,  $L_{KDE-PS}$  is lower than  $L_{Gauss}$  and the reason is that candidate bandwidth values have values greater than standard deviation and in these cases, a bandwidth value greater than std.dev. was selected as the ideal bandwidth, resulting in a lower leakage estimation. POWERBROKER 4's results show  $L_{Gauss}$  is overestimating the leakage but std.dev. is actually 100 times lower than our fixed bandwidth, resulting in  $L_{KDE-Fixed}$  overfitting on the data but the fixed methods still overestimate the leakage, reporting 100% leakage on other features.

Comparing Leakiest to KDE-ParamSearch, Leakiest sometimes underestimates the leakage (SNAPBUDDY) and it is unable to produce a result when the number of samples is too low (GABFEED). Leakage quantification took between 45 minutes and 5.5 hours on almost all applications and exact runtime per application can be seen on

Table 2. Only TOURPLANNER takes more than a day to analyze in total. The reason is size of the secret domain of TOURPLANNER is much greater than other applications, at least 6 times more, and the parameter search is done to estimate  $p(x|s)$  for each secret value  $s$  in the secret domain  $S$ , making the runtime proportional with size of the secret domain.

In summary, KDE-ParamSearch, with a stop criterion, converges to a leakage value between Histogram and Gaussian, in most cases greater than the lower bound identified by F-BLEAU, and handles all data distributions automatically.

**Mutator weighing comparison.** To test the effectiveness of assigning weights to mutators, we ran all five examples starting from the same seed set, once with mutation weighing, once without mutation weighing, and estimated the leakage using KDE-ParamSearch. The goal is to see if selecting useful mutators gets the leakage results closer to the ground truth. Results are shown in Figure 2. First four cases had 62 mutators to change the secret and other variables. The fifth example has 110 mutators to change the input list: add/remove elements, shuffle characters, replace words, shuffle list, etc.

For Example 1, leakage difference between two runs is minimal because the observable value does not depend on the input. For Examples 2–5, the run with mutation weighing is able to converge faster because it gives more weight to mutators that change the parameters like  $s, x, y$  that affect the observables.

We ran a similar test on some STAC apps with complex inputs like RAILYARD and there is some difference between leakage results with and without mutator weighing (for top feature, 28% without weighing, 22% with weighing) but without the ground truth, it is impossible to evaluate if our approach improved the leakage estimation on the STAC apps.

**Automated input set generation.** Results in Figure 1 also show one of the key advantages. Consider the leakage values computed on Example 2. A tool that relies on manually constructed input sets cannot differentiate the input set with 10000 inputs from the one with size 20000. However, the leakage values for these input sets are very different. Based on its feedback-driven iterative approach, AutoFeed is able to converge to an accurate leakage estimation automatically starting from the same input set.

**Leakage results for different noise levels.** To demonstrate that AutoFeed also produces meaningful results on a noisy network environment, we simulate the same experiments as if the servers are on three different locations. We measured the latency of three servers, one in US West Coast (Google servers, latency: 3.43 ms avg, 0.08 ms std.dev), one in US East Coast (Wikimedia servers, latency: 74.64 ms avg, 3.20 ms std.dev), and one in Russia (VK servers, latency: 220.52 ms avg, 2.38 ms std.dev). We used these latency values to add Gaussian timing noise to the obtained packets and simulate a noisy network environment. These simulations only affect the cases where we look for timing side channels. The results are in Table 3. We expect the leakages to drop because of extra collisions created by noisy environments. For all cases, the leakages drop when compared to the original experiments as we predicted. Some cases like GABFEED 1 are affected less than others. We believe this is because there is not much overlap between the distributions and the separation is greater than the level of noise.

**Table 2: Leakage results using AutoFeed with different probability estimation methods.**  $L_{Gauss}$  and  $L_{Hist}$  are Gaussian-based and histogram-based estimations respectively.  $L_{Leakiest}$  is Leakiest-based estimation.  $L_{KDE-Fix}$ ,  $L_{KDE-SD}$ ,  $L_{KDE-PS}$  are using KDE with a fixed bandwidth, standard deviation based bandwidth and parameter search based bandwidth respectively.  $L_{F-BLEAU}^*$  is min-entropy results using the F-BLEAU tool. Top Feature is the top feature when run with  $L_{KDE-PS}$ . Runtime describes total analysis runtime in minutes.

Programs	Type	Vulnerability	Top Feature-AutoFeed	$L_{Gauss}$	$L_{Hist}$	$L_{Leakiest}$	$L_{F-BLEAU}^*$	$L_{KDE-SD}$	$L_{KDE-Fix}$	$L_{KDE-PS}$	Iter.	Runtime
AIRPLAN 2	Space	Present	$\Sigma$ Sizes Phase 4 ↓	100%	100%	99%	90%	100%	100%	100%	3	76 min.
AIRPLAN 5	Space	Mitigated	$\Sigma$ Sizes Phase 4 ↓	89%	94%	74%	82%	88%	93%	89%	4	114 min.
AIRPLAN 3	Space	Absent	Size Pkt 20 ↓	46%	33%	21%	20%	45%	35%	47%	6	161 min.
RAILYARD	Space	Absent	Size Pkt 2 ↓	22%	27%	21%	27%	20%	22%	22%	12	202 min.
SNAPBUDDY	Space	Present	$\Sigma$ Sizes Full Trace ↑	100%	100%	47%	100%	100%	100%	100%	3	47 min.
GABFEED 1	Time	Present	$\Delta$ Pkt 12-13 ↓	99%	100%	N/A	60%	100%	100%	98%	8	108 min.
GABFEED 2	Time	Absent	$\Delta$ Pkt 11-12 ↓	29%	71%	N/A	24%	31%	66%	31%	19	297 min.
GABFEED 5	Time	Absent	$\Delta$ Pkt 11-12 ↓	29%	65%	N/A	21%	31%	61%	32%	15	240 min.
POWERBROKER 1	Time	Present	$\Delta$ Pkt 9-10 ↑	43%	100%	42%	53%	45%	100%	39%	20	313 min.
POWERBROKER 2	Time	Absent	$\Delta$ Pkt 43-44 ↓	11%	32%	3%	18%	10%	24%	15%	18	263 min.
POWERBROKER 4	Time	Absent	$\Delta$ Pkt 28-29 ↓	22%	9%	9%	32%	26%	9%	25%	16	220 min.
BIDPAL 2	Time	Present	$\Delta$ Pkt 28-29 ↓	22%	100%	33%	32%	23%	100%	23%	17	217 min.
BIDPAL 1	Time	Absent	$\Delta$ Pkt 35-36 ↓	2%	39%	3%	15%	3%	35%	14%	10	137 min.
TOURPLANNER	Time	Present	$\Delta$ Pkt 12-13 ↓	60%	70%	62%	42%	62%	67%	60%	12	2057 min.

**Table 3: Leakage results using  $L_{KDE-PS}$  for four different noise conditions.**

Programs	Type	Vulnerability	Top Feature-AutoFeed	STAC Platform	US-West	US-East	Russia
GABFEED 1	Time	Present	$\Delta$ Pkt 12-13 ↓	98%	97%	96%	96%
GABFEED 2	Time	Absent	$\Delta$ Pkt 11-12 ↓	31%	29%	9%	8%
GABFEED 5	Time	Absent	$\Delta$ Pkt 11-12 ↓	32%	23%	8%	7%
POWERBROKER 1	Time	Present	$\Delta$ Pkt 9-10 ↑	39%	39%	37%	36%
POWERBROKER 2	Time	Absent	$\Delta$ Pkt 43-44 ↓	15%	14%	3%	3%
POWERBROKER 4	Time	Absent	$\Delta$ Pkt 28-29 ↓	25%	20%	9%	8%
BIDPAL 2	Time	Present	$\Delta$ Pkt 28-29 ↓	23%	23%	22%	23%
BIDPAL 1	Time	Absent	$\Delta$ Pkt 35-36 ↓	14%	9%	8%	3%
TOURPLANNER	Time	Present	$\Delta$ Pkt 12-13 ↓	60%	50%	5%	5%

## 6 RELATED WORK

Profit [40] is a black-box tool to detect and quantify side channels in network traffic. The user must provide the set of inputs to run. This makes the tool impractical. Different input suites yield different results, and the tool offers no way to assess their reliability. An input suite too small or skewed yields incorrect results; an input suite too large and diverse may entail immense wasted effort, making the analysis cost prohibitive. Striking a balance between an insufficient input suite and a wasteful one is tedious, costly, and problem-specific. In contrast, AutoFeed automates this process.

Chen et al. [13] study side-channel leaks in Web applications using a stateful model that relates transitions between system states to side-channel observables. They show vulnerabilities and look into mitigation costs. They do not provide a tool or quantify leakage. Chapman and Evans [10] present a technique for black-box side channel detection in Web applications by crawling the application and building an automaton. They associate transitions between app states with captured network traffic, and build classifiers to recognize, on future traffic, which transition is likely to have been triggered. They use the Fisher criterion [19] to quantify information leakage based on distinguishability of data points. They use simpler aggregate features, like total size difference or edit distance.

Privacy Oracle [24] finds leaks using differential testing. Like AutoFeed, it is black-box, and it uses alignment to detect meaningful relationships across network traces. But it assumes that network traffic is unencrypted. AutoFeed does not rely on such an assumption: it exploits publicly observable side-channel metadata.

AppScanner [47] is a tool for identifying different apps from encrypted network traces. It is black-box and trains classifiers on traces which can identify which app is being used. They focus on a single type of secret, whereas AutoFeed is a more general tool.

F-BLEAU [14] is a black-box side channel detection and quantification tool that uses  $k$ -nearest neighbors estimation to generalize the estimation to unseen data, and min-entropy to quantify information leakage. Leakiest [15] is a tool for side channel detection that uses models based on histograms and KDE, with bandwidth based on std.dev. It provides confidence intervals, but only if there was enough data, which cannot be known until after the analysis.

None of the aforementioned black-box tools offer automated input generation. As a consequence, none of them can offer *adaptive* input generation. AutoFeed provides dynamic, adaptive input generation, using feedback-driven self-adjustment to reduce wasted effort and improve the quality of the results obtained.

Of the related work that addresses the problem of software side-channel detection, a significant portion are white-box techniques and tools. As mentioned in Section 1, white-box tools require access to the source code of the target system, and cannot handle systems written in multiple languages or that involve multiple components.

DiffFuzz [32] is a side-channel analysis technique based on differential fuzzing. Like AutoFeed, it involves a feedback loop, but it is white-box. Its evaluation uses manually sliced parts of programs, where crucial classes or methods relevant to the side channel are manually isolated and compiled together with the tool as a program. AutoFeed can analyze unmodified systems, and since it interacts

with them at the network level, it can analyze systems written in any language or combination thereof. Other key differences are that AutoFeed handles noise and nondeterminism while DiffFuzz assumes determinism and precise measurements, and that AutoFeed quantifies the amount of information leaked.

CoCoChannel [8] analyzes the control flow graph of the system with respect to an execution cost model. Given a secret of interest, it builds symbolic cost expressions and reduces detecting imbalances to constraint solving. Themis [11] is an end-to-end static analysis tool for side-channel analysis of Java code based on Quantitative Cartesian Hoare Logic. Blazer [4] is a static program analysis tool that can prove the absence of timing side channels by decomposition. Scanner [12] is a static analysis tool for side-channel vulnerability detection in PHP-based Web applications. All of the above are white-box analysis tools that depend on source code, and suffer from the aforementioned limitations.

Several works use symbolic execution to quantify information leakage statically [22, 37–39]. They combine symbolic execution with model counting or quantitative information flow. All of these are white-box and require source code. Furthermore, their scalability is limited by that of symbolic execution. They are also limited to the analysis of systems written in a single language.

Fuzzing techniques are popular in security testing. Coverage-guided fuzzing [29, 42, 49] can generate complex, structured inputs. Many fuzzing engines use mutation. Some frameworks allow for custom mutators [35]. Others combine fuzzing with symbolic execution [33, 46]. However, coverage-guided fuzzers depend on code instrumentation, and thus require source code. Also, fuzzing engines are generally built toward the goal of breaking the system—that is, finding inputs that cause crashes or assertion violations, rather than quantifying leakage. Fuzzing engines also tend to assume that it is possible to execute the system in milliseconds, while AutoFeed deals with systems that can take many seconds per input.

Work by Bang et al. [5] performs online synthesis of adaptive side-channel attacks. It uses another kind of feedback loop. Like AutoFeed, it profiles the program through the network. However, it is still a white-box technique due to its need to symbolically execute the program before running it. Due to its dependency on symbolic execution, it cannot handle large systems.

## 7 CONCLUSIONS

We presented AutoFeed, a black-box tool to detect and quantify side-channel information leakage in networked software systems. AutoFeed significantly reduces the manual effort required by prior black-box side-channel analysis approaches by providing a feedback-driven automated process for input space exploration and information leakage estimation. Given a set of input mutators and a small number of seed inputs, AutoFeed iteratively mutates inputs and periodically updates its leakage estimations to identify the features that leak most information about the secret. AutoFeed measures the effect of different mutator subsets on leakage, and assigns weights to prioritize mutators that produce more changes in the leakage estimation. AutoFeed uses kernel density estimation and an automated search mechanism to determine crucial hyperparameter values, in order to estimate probability distributions for modeling the observed data. It uses a stop criterion to detect convergence of

the leakage estimation and terminate the analysis. Our experimental evaluation on the benchmarks shows that AutoFeed is effective in automatically detecting and quantifying information leaks.

## REFERENCES

- [1] [n.d.] *Docker API for Python*.
- [2] [n.d.] *Numpy: scientific computing with Python*.
- [3] Onur Aciicmez. 2007. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*. ACM, 11–18.
- [4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Teruchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017*. 362–375. <https://doi.org/10.1145/3062341.3062378>
- [5] Lucas Bang, Nicolas Rosner, and Tevfik Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24–26, 2018*. 307–322. <https://doi.org/10.1109/EuroSP.2018.00029>
- [6] Philippe Biondi. [n.d.] *Scapy: Packet crafting for Python*.
- [7] Adrian W Bowman. 1984. An alternative method of cross-validation for the smoothing of density estimates. *Biometrika* 71, 2 (1984), 353–360.
- [8] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16–21, 2018*. 27–37. <https://doi.org/10.1145/3213846.3213867>
- [9] Prabin Burman. 1989. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika* 76, 3 (1989), 503–514.
- [10] Peter Chapman and David Evans. 2011. Automated Black-box Detection of Side-channel Vulnerabilities in Web Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (Chicago, Illinois, USA) (CCS '11)*. ACM, New York, NY, USA, 263–274. <https://doi.org/10.1145/2046707.2046737>
- [11] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 875–890. <https://doi.org/10.1145/3133956.3134058>
- [12] Jia Chen, Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2017. Static Detection of Asymptotic Resource Side-channel Vulnerabilities in Web Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (Urbana-Champaign, IL, USA) (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 229–239. <http://dl.acm.org/citation.cfm?id=3155562.3155595>
- [13] Shuo Chen, Kehuan Zhang, Rui Wang, and Xiaofeng Wang. 2010. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. *2010 IEEE Symposium on Security and Privacy (SP) 00* (2010), 191–206. <https://doi.org/doi.ieeecomputersociety.org/10.1109/SP.2010.20>
- [14] Giovanni Cherubin, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2019. F-BLEAU: Fast Black-box Leakage Estimation. *CoRR* abs/1902.01350 (2019). arXiv:1902.01350 <http://arxiv.org/abs/1902.01350>
- [15] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 690–695. [https://doi.org/10.1007/978-3-642-39799-8\\_47](https://doi.org/10.1007/978-3-642-39799-8_47)
- [16] DARPA. 2015. *The Space-Time Analysis for Cybersecurity (STAC) program*. <http://www.darpa.mil/program/space-time-analysis-for-cybersecurity>
- [17] DARPA. 2017. *Public release items for the DARPA Space-Time Analysis for Cybersecurity (STAC) program*. <https://github.com/Apogee-Research/STAC>
- [18] Inc. Docker. [n.d.] Docker. <https://www.docker.com/>
- [19] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics* 7, 2 (1936), 179–188.
- [20] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14–16, 2001, Proceedings*. 251–261. [https://doi.org/10.1007/3-540-44709-1\\_21](https://doi.org/10.1007/3-540-44709-1_21)
- [21] Peter Hall, JS Marron, and Byeong U Park. 1992. Smoothed cross-validation. *Probability theory and related fields* 92, 1 (1992), 1–20.
- [22] Xujing Huang and Pasquale Malacaria. 2013. SideAuto: quantitative information flow for side-channel leakage in web applications. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society, WPES 2013, Berlin, Germany, November 4, 2013*. 285–290. <https://doi.org/10.1145/2517840.2517869>
- [23] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering* 9, 3 (2007), 90–95. <https://doi.org/10.1109/MCSE.2007.55>

- [24] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) (CCS '08). ACM, New York, NY, USA, 279–288. <https://doi.org/10.1145/1455770.1455806>
- [25] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*. Springer, 97–110.
- [26] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 <http://arxiv.org/abs/1801.01203>
- [27] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.
- [28] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. 388–397. [https://doi.org/10.1007/3-540-48405-1\\_25](https://doi.org/10.1007/3-540-48405-1_25)
- [29] lcamtuf. [n.d.]. American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl/>
- [30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 973–990. <https://www.usenix.org/conference/usenixsecurity18/presentation/lipp>
- [31] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of Power Analysis Attacks on Smartcards. *Smartcard* 99 (1999), 151–161.
- [32] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2018. DiffFuzz: Differential Fuzzing for Side-Channel Analysis. *CoRR* abs/1811.07005 (2018). arXiv:1811.07005 <http://arxiv.org/abs/1811.07005>
- [33] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) (ISSTA 2018). ACM, New York, NY, USA, 322–332. <https://doi.org/10.1145/3213846.3213868>
- [34] Emanuel Parzen. 1962. On Estimation of a Probability Density Function and Mode. *Ann. Math. Statist.* 33, 3 (09 1962), 1065–1076. <https://doi.org/10.1214/aoms/117704472>
- [35] PeachTech. [n.d.]. PeachFuzzer. <http://www.peach.tech/>
- [36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [37] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 328–342. <https://doi.org/10.1109/CSF.2017.8>
- [38] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo d'Amorim. 2014. Quantifying information leaks using reliability analysis. In *2014 International Symposium on Model Checking of Software, SPIN 2014, Proceedings, San Jose, CA, USA, July 21-23, 2014*. 105–108. <https://doi.org/10.1145/2632362.2632367>
- [39] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. 2012. Symbolic Quantitative Information Flow. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5. <https://doi.org/10.1145/2382756.2382791>
- [40] Nicolás Rosner, İsmet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2019. Profit: Detecting and Quantifying Side Channels in Networked Applications. In *26th Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*.
- [41] Mats Rudemo. 1982. Empirical choice of histograms and kernel density estimators. *Scandinavian Journal of Statistics* (1982), 65–78.
- [42] K Serebryany. 2015. libFuzzer, a library for coverage-guided fuzz testing. *LLVM project* (2015).
- [43] Claude E Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948), 379–423.
- [44] Bernard W. Silverman. 1986. *Density Estimation for Statistics and Data Analysis*. Springer. <https://doi.org/10.1007/978-1-4899-3324-9>
- [45] Dawn Xiaodong Song, David A. Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*. <http://www.usenix.org/publications/library/proceedings/sec01/song.html>
- [46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.
- [47] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. 2018. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security* 13, 1 (Jan 2018), 63–78. <https://doi.org/10.1109/TIFS.2017.2737970>
- [48] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*, Vol. 1. 22–25.
- [49] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. Generating Software Tests. In *Generating Software Tests*. Saarland University. <https://www.fuzzingbook.org/> Retrieved 2019-01-14 00:29:35-08:00.
- [50] Walter Zucchini, A Berzel, and O Nenadic. 2003. Applied smoothing techniques. *Part I: Kernel Density Estimation* 15 (2003).

# Scalable Analysis of Interaction Threats in IoT Systems

Mohannad Alhanahnah<sup>\*</sup><sup>†</sup>

mohannad@huskers.unl.edu

University of Nebraska–Lincoln

Computer Science and Engineering

Lincoln, Nebraska, USA

Clay Stevens<sup>\*</sup>

clay.stevens@huskers.unl.edu

University of Nebraska–Lincoln

Computer Science and Engineering

Lincoln, Nebraska, USA

Hamid Bagheri

bagheri@unl.edu

University of Nebraska–Lincoln

Computer Science and Engineering

Lincoln, Nebraska, USA

## ABSTRACT

The ubiquity of Internet of Things (IoT) and our growing reliance on IoT apps are leaving us more vulnerable to safety and security threats than ever before. Many of these threats are manifested at the interaction level, where undesired or malicious coordinations between apps and physical devices can lead to intricate safety and security issues. This paper presents IoTCom, an approach to automatically discover such hidden and unsafe interaction threats in a compositional and scalable fashion. It is backed with automated program analysis and formally rigorous violation detection engines. IoTCom relies on program analysis to automatically infer the relevant app’s behavior. Leveraging a novel strategy to trim the extracted app’s behavior prior to translating them to analyzable formal specifications, IoTCom mitigates the state explosion associated with formal analysis. Our experiments with numerous bundles of real-world IoT apps have corroborated IoTCom’s ability to effectively detect a broad spectrum of interaction threats triggered through cyber and physical channels, many of which were previously unknown, and to significantly outperform the existing techniques in terms of scalability.

## CCS CONCEPTS

• Security and privacy → Software and application security.

## KEYWORDS

Interaction Threats, IoT Safety, Formal Verification

### ACM Reference Format:

Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. 2020. Scalable Analysis of Interaction Threats in IoT Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397347>

<sup>\*</sup>Equal contribution

<sup>†</sup>Now at University of Wisconsin-Madison.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA ’20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397347>

## 1 INTRODUCTION

Internet-of-Things (IoT) ecosystems are becoming increasingly widespread, particularly in the context of the smart home. Industry forecasts suggest the average smart home will end the current year with as many as 50 connected devices [32]. The race to configure, control, and monitor these devices produces a web of different platforms all operating within the same cyber-physical environment. These platforms also allow users to install third-party software *apps*, which can intricately interact with each other, allowing complex and varied automations. Such diversity enhances the user’s experience by delivering many options for automating their home, but it comes at a price; it also escalates the attack surface for safety and security threats. The increased complexity produced by the coordination and interaction of these apps subjects the system to the risk of undesired behavior, whether by misconfiguration, developer error, or malice. For example, an app that unlocks the door when a user returns home may be subverted—either accidentally or intentionally—to unlock the door when the user is not actually present. This risk is exacerbated by the unpredictable nature of IoT environments, as it is not known *a priori* which apps and devices will be installed in tandem. The increased impact of these physical risks makes identification of risky interactions even more important.

In this context, the safety implications and security risks of IoTs have been a thriving subject of research for the past few years [3, 15, 17, 19, 20, 24, 26, 33, 38, 39, 47, 57, 58, 64]. These research efforts have scrutinized deficiencies from various perspectives. However, existing detection techniques target only certain types of inter-app threats [20, 38, 47] and do not take into account *physical channels*, which can underpin risky interactions among apps. Moreover, the state-of-the-art techniques suffer from acknowledged missing of interaction threats, as they require manual specification of the initial configuration for each app to be analyzed. This, in turn, results in missing potentially unsafe behavior if it appears from different configurations [20, 26, 38, 47]. Additionally, these analyses have been shown to experience scalability problems when applied on large numbers of IoT apps [20, 38, 47].

To address this state of affairs, this paper presents a novel approach and accompanying tool suite, called IoTCom, for compositional analysis of such hidden and unsafe interaction threats in a given bundle of cyber and physical components co-located in an IoT environment. IoTCom first utilizes a path-sensitive static analysis to automatically generate an inter-procedural control flow graph (ICFG) for each app. It then applies a novel graph abstraction technique to model the behavior relevant to the devices connected to the app as a *behavioral rule graph* (BRG), which derives rules

from IoT apps via linking the triggers, actions, and logical conditions of each control flow in each app. Unlike prior techniques, our approach respects the conditions along each branch rather than only the triggers and actions. IoTCom then automatically generates formal app specifications from the BRG models. Lastly, it uses a lightweight formal analyzer [37] to check bundles of those models for violations of multiple safety and security properties arising from interactions among the apps rules.

IoTCom has several advantages over existing work. First, unlike prior work, IoTCom supports the detection of violations occurred through physically mediated interactions by explicitly modeling a mapping of each device capability to the pertinent physical channels. Second, while prior approaches require manual specification of the initial configuration, IoTCom exhaustively identifies all initial configurations, which in turn enables automatically checking them for potential interaction violations with no need for any manual configuration. Third, our novel BRG abstraction technique optimizes the performance of our analysis and markedly improves our scalability over state-of-the-art techniques by effectively trimming the automatically-extracted ICFGs, eliding all nodes and edges irrelevant to the app's behavior from the analysis.

Using a prototype implementation of IoTCom, we evaluated its performance in detecting prominent classes of IoT coordination threats among thousands of publicly-available real-world IoT apps developed using diverse technologies. Our results corroborate IoTCom's ability to effectively detect complex coordinations among apps communicating via both cyber and physical means, many of which were previously unreported. We also demonstrate IoTCom's significantly improved scalability when compared to existing IoT threat detection techniques.

We further compare the precision of IoTCom to the other approaches using a set of benchmark IoT apps, developed by the other research groups [20, 26]. IoTCom is up to 68.8% more successful in detecting safety violations. Additionally, compared to the state-of-the art techniques in detecting safety and security violations in IoT environments, IoTCom reduces the violation detection time by 92.1% on average and by as much as 99.5%. To summarize, this paper makes the following contributions:

- *Classification of interaction threats between IoT apps.* We identify and rigorously define seven classes of multi-app interaction threats between IoT apps over physical and cyber channels both within and among apps.
- *Formal model of IoT systems.* We develop a formal specification of IoT systems, respecting *cyber and physical channels* and representing the behavior of IoT apps apropos the detection of safety and security vulnerabilities. We construct this specification as a reusable Alloy [1] module to which all extracted app models conform.
- *Automated analysis.* We show how to exploit the power of our formal abstractions by building a modular model extractor that uses static analysis techniques to automatically extract the precise behavior of IoT apps into a trimmed *behavioral rule graph*, respecting the *logical conditions* that impact the behavior of the app rules, which is then captured in a format amenable to formal analysis.

- *Experiments.* We evaluate the performance of IoTCom against real-world apps developed for *multiple IoT platforms* (SmartThings and IFTTT), corroborating IoTCom's ability in effective compositional analysis of IoT apps interaction vulnerabilities in the order of minutes. We make research artifacts, including the entire Alloy specifications, and the experimental data available to the research and education community [35].

## 2 BACKGROUND AND MOTIVATION

Smart home IoT platforms are cyber-physical systems comprising both virtual elements, such as software, and physical devices, like sensors or actuators. In popular smart home platforms, such as SmartThings [49], Apple's HomeKit [5], GoogleHome [29], Zapier [63] and MicrosoftFlow [44], physical devices installed in the home are registered with virtual proxies in a cloud-based backend. Each proxy tracks the state of the device via one or more *attributes*, which can assume different *values*. The backend also allows the user to install software *apps*, which automate the activities of these devices by applying custom *rules* that act on the virtual proxies. These rules adopt a *trigger-condition-action* paradigm:

**Triggers:** Cyber or physical events reported to the smart home system by the devices, such as a motion sensor being activated, trigger the rules.

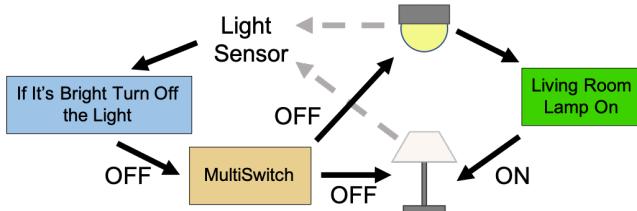
**Conditions:** Logical predicates defined on the current state of the devices determine if the rule should execute. For example, a rule might only execute if the system is in "home" mode.

**Actions:** If the conditions are met, the rule changes the state of one or more devices, which could result in a physical change like activating a light switch.

The safety and security of these systems is a major concern [17, 38, 47], particularly regarding software apps. Users can install multiple, arbitrary apps which can interact not only with physical devices in the smart home, but also with each other. *Multi-app coordination threats among smart home apps arise when two or more app rules interact to produce a surprising, unintended, or even dangerous result in the physical environment of the smart home.* Apps can interact over *cyber channels* such as shared device proxies, global settings, or scheduled tasks. We refer to coordination over cyber channels as *direct coordination*. They also interact over *physical channels* [26] via a shared metric acted upon by an *actuator* and monitored by a *sensor*; this is termed *indirect coordination*.

To make the idea concrete, consider three publicly-available apps, i.e., the SmartThings app *MultiSwitch* and the IFTTT automations *If It's Bright Turn Off the Light* and *Living Room Lamp On*. Figure 1 shows an example configuration of these apps along with their devices, which can enter into an infinite loop. *If It's Bright* watches a light sensor. When the light reaches a user-defined level, it turns off the *MultiSwitch*. *MultiSwitch* forwards that command to both the overhead light and the lamp. *Living Room Lamp On* responds to the overhead light going off by turning on the lamp. That, in turn, activates the light sensor through luminance physical channel, which initiates the same chain of events again.

The above example points to one of the most demanding issues in the smart home IoT ecosystem, i.e., detection of multi-app coordination threats. What is required is a system-wide reasoning—that determines how those individual rules could impact one another



**Figure 1: Example infinite actuation loop.** The apps turn the living room lights on and off again repeatedly due to a mis-configuration.

when the corresponding apps are installed together—not comfortably attainable through conventional analysis methods such as static analysis or testing, which are more suited for identifying issues in individual parts of the system. In the next sections, we first provide a classification of various multi-app coordination threats, and then present a formal modeling approach to address these issues.

### 3 MULTI-APP COORDINATION THREATS

In this section, we present seven classes of potential multi-app coordination threats that can arise due to interactions between IoT app rules. As defined earlier, a *rule* comprises a set of *triggers*, *conditions*, and *actions*. More formally, an app rule  $\rho$  is a tuple  $\rho = \langle T_\rho, C_\rho, A_\rho \rangle$ , where  $T_\rho$ ,  $C_\rho$ , and  $A_\rho$  are sets of triggers, conditions, and actions for rule  $\rho$ , respectively. Each *component* (trigger, condition, or action) can be described as a triple of a device, an attribute, and a set of values. The sets of devices, attributes, and values associated with a component  $\alpha$  are denoted as  $\mathbb{D}(\alpha)$ ,  $\mathbb{A}(\alpha)$ , and  $\mathbb{V}(\alpha)$ , respectively.

#### 3.1 Direct Coordination

Two rules  $R_i$  and  $R_j$  directly coordinate if an *action* from  $R_i$  influences the devices and attribute associated with a component of  $R_j$ . As they act on a shared physical environment, co-located apps can coordinate via both cyber and physical means. In cyber coordination, the devices and attribute of an action of  $R_i$  must *match* those referenced by some component of  $R_j$ :

*Definition 3.1 (match).* The relation *match* defines an intersection between the devices and attribute of a component  $\alpha$  from an IoT app rule  $R_i$  and a component  $\beta$  from another IoT app rule  $R_j$ :

$$\text{match}(\alpha, \beta) \equiv (\mathbb{D}(\alpha) \cap \mathbb{D}(\beta) \neq \emptyset) \wedge (\mathbb{A}(\alpha) = \mathbb{A}(\beta))$$

Physical coordination is mediated by some physical *channel*. We define the set of physical channels as  $\text{CHANNELS}$ , with  $\text{ACTUATORS}(\gamma)$  and  $\text{SENSORS}(\gamma)$  denoting the sets of devices that can either act upon or sense changes to a channel  $\gamma \in \text{CHANNELS}$ , respectively.

*Definition 3.2 (same\_channel).* The relation *same\_channel* defines physically-mediated interaction via channel  $\gamma$  between the devices of an action  $\alpha$  from an IoT app rule  $R_i$  and a trigger  $\beta$  from another IoT app rule  $R_j$ :

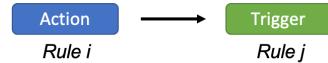
$$\begin{aligned} \text{same\_channel}(\alpha, \beta) \equiv & \exists \gamma \in \text{CHANNELS}. ((\mathbb{D}(\alpha) \subseteq \text{ACTUATORS}(\gamma)) \\ & \wedge (\mathbb{D}(\beta) \cap \text{SENSORS}(\gamma) \neq \emptyset)) \end{aligned}$$

The most common type of coordination—which is a core feature of IoT automation systems—is (*T1*) *action-trigger* coordination,

where the value of one of  $R_i$ 's actions matches a value in or actuates a channel sensed by one of  $R_j$ 's triggers, as depicted in Figure 2. This coordination is often intended but can also lead to an unintended activation of subsequent rules if misused.

*Definition 3.3 ((T1) Action-trigger).* Action  $a$  of rule  $R_i$  activates trigger  $t$  of rule  $R_j$  either directly (by involving overlapping devices, attributes, and values) or mediated by some physical channel actuated by the devices of  $a$  and sensed by the devices of  $t$ .

$$\begin{aligned} T1(R_i, R_j) \equiv & ((\forall a \in A_{R_i}, c \in C_{R_j}. (\text{match}(a, c) \Rightarrow (\mathbb{V}(a) \subseteq \mathbb{V}(c)))) \\ & \wedge (\exists a \in A_{R_i}, t \in T_{R_j}. ((\text{match}(a, t) \wedge (\mathbb{V}(a) \subseteq \mathbb{V}(t))) \\ & \vee (\text{same\_channel}(a, t)))))) \end{aligned}$$



**Figure 2: (T1) Action-trigger**

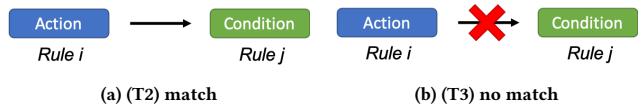
*Action-condition* coordination may be less obvious to the user; in this case,  $R_i$  either (*T2*) enables or (*T3*) disables  $R_j$  depending on whether or not the values of the action and the related condition match, as shown in Figures 3a and 3b.

*Definition 3.4 ((T2) Action-condition (match)).* Rule  $R_i$  executes action  $a$  that changes the system to satisfy condition  $c$  of rule  $R_j$ .

$$T2(R_i, R_j) \equiv (\forall a \in A_{R_i}, c \in C_{R_j}. (\text{match}(a, c) \Rightarrow (\mathbb{V}(a) \subseteq \mathbb{V}(c))))$$

*Definition 3.5 ((T3) Action-condition (no match)).* Rule  $R_i$  executes action  $a$  that changes the system to no longer satisfy condition  $c$  of rule  $R_j$ .

$$T3(R_i, R_j) \equiv (\exists a \in A_{R_i}, c \in C_{R_j}. (\text{match}(a, c) \wedge (\mathbb{V}(a) \cap \mathbb{V}(c) = \emptyset)))$$



**Figure 3: Action-condition**

#### 3.2 Chain Coordination

The first three classes define ways for two rules to coordinate directly, but apps may also coordinate via a chain of rules—each coordinating with another via action-trigger (*T1*) coordination—or via a relationship between their triggering event(s). We refer to two rules that can be triggered by the same event as *siblings* if their conditions are not mutually exclusive.

*Definition 3.6 (sibling).* Relation *sibling* holds between two rules  $R_i$  and  $R_j$  if and only if there is a trigger  $t_1$  in  $R_i$  that overlaps via devices, attributes, and values with a trigger  $t_2$  in  $R_j$ ; none of the conditions in  $R_i$  violate any condition of  $R_j$ ; and vice versa.

$$\begin{aligned} \text{sibling}(R_i, R_j) \equiv & ((\forall c_1 \in C_{R_i}, c_2 \in C_{R_j}. \\ & (\text{match}(c_1, c_2) \Rightarrow (\mathbb{V}(c_1) \cap \mathbb{V}(c_2) \neq \emptyset))) \\ & \wedge (\exists t_1 \in T_{R_i}, t_2 \in T_{R_j}. (\text{match}(t_1, t_2) \wedge (\mathbb{V}(t_1) \cap \mathbb{V}(t_2) \neq \emptyset)))) \end{aligned}$$

Chain coordination may lead to the scenario where the action of a chain of rules (or a single rule itself) triggers one of the rules previously involved in the same chain. We define this class of multi-app coordination threats as *(T4) self coordination* (shown in Figure 4):

*Definition 3.7 ((T4) Self coordination).* Rule  $R_i$ —either directly or through a transitive chain of intermediate rules connected by the  $T_1$  relation, denoted  $T_1^+$ —triggers itself.

$$T4(R_i) \equiv T_1^+(R_i, R_i)$$

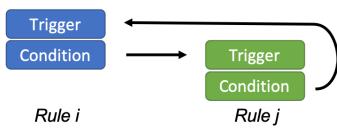


Figure 4: (T4) Self coordination

*Action-action* coordination occurs when two distinct rules act upon the same attribute of the same device. If the rules are triggered by unrelated events, there is no coordination between the two rules. If the triggering events are the same, then the rules may coordinate to produce an undesired result such as a race condition or additional wear on a given device (cf. Figures 5a-5b):

*Definition 3.8 ((T5) Action-action (conflict)).* Two rules  $R_i$  and  $R_j$  that are each triggered by the same event—either directly or through a chain of coordinating rules—each has an action with the same device and attribute but different values. ( $T_1^*$  denotes the reflexive transitive closure of the  $T_1$  relation).

$$\begin{aligned} T5(R_i, R_j) \equiv & (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R. (match(a_1, a_2) \\ & \wedge (\forall (a_1) \neq \forall (a_2)) \wedge T_1^*(R_m, R_i) \wedge T_1^*(R_n, R_j) \\ & \wedge ((R_m = R_n) \vee sibling(R_m, R_n))) \end{aligned}$$

*Definition 3.9 ((T6) Action-action (repeat)).* Two rules  $R_i$  and  $R_j$  that are each triggered by the same event—either directly or through a chain of coordinating rules—each has an action with the same device, attribute, and value.

$$\begin{aligned} T6(R_i, R_j) \equiv & (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R. (match(a_1, a_2) \\ & \wedge (\forall (a_1) = \forall (a_2)) \wedge T_1^*(R_m, R_i) \wedge T_1^*(R_n, R_j) \\ & \wedge ((R_m = R_n) \vee sibling(R_m, R_n))) \end{aligned}$$

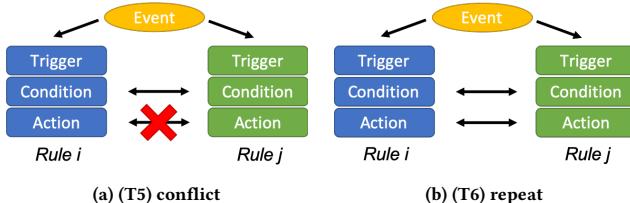


Figure 5: Action-action

The prior chain coordinations result from a single initiating event; it is also possible for rules to be configured such that a given outcome is guaranteed even in the face of mutually-exclusive events. We define a seventh class of coordination—*(T7) exclusive event coordination*—to detect this case, shown in Figure 6.

*Definition 3.10 ((T7) Exclusive event coordination).* Two rules  $R_i$  and  $R_j$  that would be triggered by *mutually exclusive* events—which share a device and attribute but different values—have actions that share the same device, attribute, and value.

$$\begin{aligned} T7(R_i, R_j) \equiv & (\exists a_1 \in A_{R_i}; a_2 \in A_{R_j}; R_m, R_n \in R; t_1 \in T_{R_m}; t_2 \in T_{R_n}. \\ & (match(a_1, a_2) \wedge (\forall (a_1) = \forall (a_2)) \wedge T_1^*(R_m, R_i) \wedge T_1^*(R_n, R_j) \\ & \wedge match(t_1, t_2) \wedge (\forall (t_1) \cap \forall (t_2) \neq \emptyset)) \end{aligned}$$

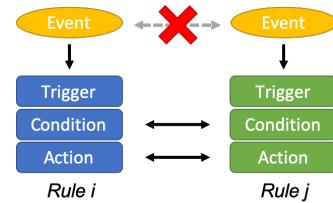


Figure 6: (T7) Exclusive event

In the rest of the paper, we show how, through a practical combination of static analysis and a lightweight formal method, IoTCom can automatically discover such unsafe and intricate interaction threats in a compositional and scalable fashion.

## 4 APPROACH OVERVIEW

This section introduces IoTCom, a technique that automatically determines whether the interactions within an IoT environment could compromise the safety and security thereof. Figure 7 illustrates the architecture of IoTCom and its two major components, described in detail in the following two sections:

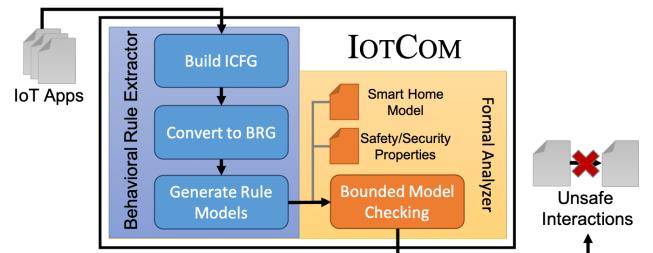


Figure 7: IoTCom System Overview.

**(1) Behavioral Rule Extractor (Section 5):** The *Behavioral Rule Extractor* component automatically infers models of the apps behavior using a novel graph abstraction technique. The component first performs static analysis on each app to generate an inter-procedural control flow graph (ICFG). It then creates a *behavioral rule graph* containing only the flows pertinent to the events and commands forwarded to/from physical IoT devices, along with any conditions required for those actions. Each flow is then automatically transformed into a formal model of the app.

**(2) Formal Analyzer (Section 6):** The *Formal Analyzer* component is then intended to use lightweight formal analysis techniques to verify specific properties (i.e., IoT coordination threats) in the extracted specifications. IoTCom uses three formal specifications:

(1) a *base model of smart home IoT systems* that defines foundational rules for cyber and physical channels, IoT apps, how they behave, and how they interact with each other, (2) assertions for *safety and security properties*, and (3) the *IoT app behavioral rule models* automatically generated by the previous component for each app. Those specifications are then checked together for detecting violations of the properties.

## 5 BEHAVIORAL RULE EXTRACTOR

The Behavioral Rule Extractor executes three main steps to automatically infer the behavior of individual IoT apps: (1) build an *inter-procedural control flow graph (ICFG)*; (2) convert the ICFG to a *behavioral rule graph (BRG)*; and (3) generate formal models for the behavioral rules.

### 5.1 Building ICFG

The Behavioral Rule Extractor first generates an inter-procedural control flow graph for each app. It analyzes the abstract syntax tree of the given app to build a *call graph* of local and API-provided methods as well as a *control flow graph* for each local method. Each graph is generated using a path-sensitive analysis [41] to preserve the logical conditions along each control flow. To capture the precise behavior of IoT apps, it is essential to extract the predicates that will influence the actions performed by IoT apps. Path-sensitivity supports this goal. Many of the triggered methods in the IoT apps are calling other methods that will perform certain actions. As such, an inter-procedural analysis is required to consider the calling context when analyzing the target of a triggered method. This, in turn, improves the precision of our approach compared to the state-of-the-art techniques, which do not consider the conditions when identifying interactions between apps [26]. The Behavioral Rule Extractor then combines each control flow graph with the call graph to construct an ICFG starting at each *entry method* in the graph. The details of generating the ICFG depend on how apps are defined for each platform.

IFTTT applets are reactive rules that interact with REST services exposed by service providers [34]. Each applet consists of a single trigger-action pair. The Behavioral Rule Extractor treats each applet as a standalone IoT app defining exactly one rule. It performs string analysis [22] to extract an ICFG comprising one entry node for the trigger and one “method call” invoking a device API for the action. For instance, IFTTT applet “If I arrive at my house then open my garage door” [28] would result in an ICFG with an entry node for “arrive at my house” and a method call node for “open my garage door”. Automatically identifying the corresponding keywords specified in triggers and actions, in turn, allows IoTCom to detect the associated capability, attribute, and value in the next steps.

SmartThings apps are written as Groovy programs, allowing for multiple rules and more extensive logic. To generate an ICFG for a SmartThings app, the Behavioral Rule Extractor first extracts principal information regarding: (1) the devices and attributes used in the app, (2) the user’s configuration of the app, (3) any global variables defined in the SmartThings documentation [49] or set using the state object, and (4) the entry methods of the app’s triggers. The SmartThings platform defines entry methods using calls to specific API methods: subscribe, schedule, runIn, and

---

### Listing 1 Groovy code for MultiSwitch app

---

```

1 preferences {
2     section("When this switch is toggled...") {
3         input "master", "capability.switch", title: "Where?" }
4     section("Turn on/off these switches...") {
5         input "switches", "capability.switch", multiple: true } }
6 def installed() {
7     subscribe(master, "switch.On", switchOn)
8     subscribe(master, "switch.Off", switchOff) }
9 def updated() {
10    unsubscribe()
11    subscribe(master, "switch.On", switchOn)
12    subscribe(master, "switch.Off", switchOff) }
13 def switchOn(evt) {
14     log.debug "Switches on"
15     switches?.on() }
16 def switchOff(evt) {
17     log.debug "Switches off"
18     switches?.off() }
```

---

runOnce. Next, the Behavioral Rule Extractor creates a control flow graph for each of those entry methods. These graphs are combined to generate an inter-procedural control flow graph for the IoT app. Note that existing state-of-the-art analysis techniques lack support for direct program analysis of Groovy code. By performing the analysis directly on the Groovy code, IoTCom avoids the pitfalls (and cost) of translating the code into some intermediate representation. As a concrete example, Listing 1 shows the Groovy code defining the *MultiSwitch* app described in Section 2. The calls to subscribe on Lines 11-12 define two entry points—switchOn and switchOff. Each of those methods would comprise two nodes in the ICFG corresponding to the logging line and API call contained in each.

### 5.2 Generating Behavioral Rule Graph

The Behavioral Rule Extractor next tailors the ICFG into a succinct, annotated graph representing the relevant behavior of the IoT app—a *behavioral rule graph (BRG)*. By eliding all edges and nodes from the ICFG that do not impact the app’s behavior with respect to physical devices, the BRG makes it easier to infer the behavior defined in the app, optimizing the performance of our analysis. To construct the BRG from the ICFG, the nodes in the ICFG are traversed starting from each entry method, generating nodes in the BRG as follows:

- **Trigger:** Entry method nodes from the ICFG are propagated to the BRG as trigger nodes.
- **Condition:** Control statements such as if blocks generate condition nodes in the BRG.
- **Action:** Any node that invokes a device API method creates an action node in the BRG.
- **Method Call:** Method calls to other local methods produce method call nodes in the BRG, as the called method may include relevant app behavior.

Continuing the example (Listing 1) from Section 5.1, the Behavioral Rule Extractor converts the ICFG for *MultiSwitch* into a BRG, starting with the entry point function switchOn. The next node in the ICFG is a call to a logging API, which has no bearing on the app’s behavior and is not included in the BRG. The node following the logging is an API call to on, which translates directly to an action node in the BRG. The resulting BRG captures all actions that affect the devices in the environment. If the example contained any

conditional statements, the BRG would also maintain the control flow reflecting the conditions for each action.

After creating the BRG, the statements corresponding to each node are converted to  $\langle \text{device}, \text{attribute}, \text{value} \rangle$  tuples. If a value in any of the nodes does not correspond directly to a member of one of those sets, we perform backward inter-procedural data flow analysis [45] to resolve the dependency.

### 5.3 Generating Rule Models

The final component of the Behavioral Rule Extractor generates formal models of each app's rules based on the BRG. As described in Section 2, the behavior of an IoT app consists of a set of rules  $R$ , where each rule is a tuple of triggers, conditions, and actions.

In order to tie the behavior of these rules back to the physical devices in the smart home, the elements of  $T$ ,  $C$ , and  $A$  are each formalized as sets of tuples of  $\langle \text{device}, \text{attribute}, \text{value} \rangle$ . Each type of device is assumed to have its own set of device-specific attributes, and each attribute constrains its own allowed values according to the device manufacturer's specifications. For example, a smart lock **device** may have a "locked" **attribute** to indicate the state of the lock, which accepts **values** of "locked" or "unlocked". An action to unlock a specific lock (**TheLock**) would contain a tuple composed of those elements, e.g.,  $\langle \text{TheLock}, \text{locked}, \text{unlocked} \rangle$ .

To generate the models from the BRG, IoTCom starts from each trigger node (which is used as the TRIGGER for the rule) and traverses the graph to find the action nodes; every rule must have at least one ACTION. From each action node, it performs a reverse depth-first search back to the trigger, collecting the tuples for each condition node encountered along the path as the CONDITIONS of the rule.

## 6 FORMAL ANALYZER

This section describes the *Formal Analyzer* component of IoTCom, which takes as input the behavioral rule models generated by the *Behavioral Rule Extractor*. These formal models are verified against various safety and security properties using a bounded model checker to exhaustively explore every interaction within a defined scope. This allows IoTCom to automatically analyze each bundle of apps without manual specification of the initial system configuration, which is required for comparable state-of-the-art techniques [20, 47]. We use Alloy [37] to demonstrate our approach for several reasons. First, it provides a concise, simple specification language suitable for declarative specification of both IoT apps and safety and security properties to be checked. In particular, Alloy includes support for modeling transitive closure, which is essential to analyze complex, chained interactions. Second, it provides a fully-automated analyzer, shown to be effective in exhaustively analyzing specifications in various domains [7–13, 43].

The bounded model checking relies on three sets of formal specifications, as shown in Figure 7: (1) a base *smart home model* describing the general entities composing a smart home environment; (2) the app-specific *behavioral rule models* generated by the *Behavioral Rule Extractor*; and (3) formal assertions for our *safety and security properties*. Complete Alloy models are available online at our project site [35].

---

### Listing 2 Excerpt of base smart home Alloy model.

---

```

1  abstract sig Device { attributes : set Attribute }
2  abstract sig Attribute { values : set Value }
3  abstract sig Value {}
4  abstract sig IoTApp { rules : set Rule }
5  abstract sig Rule {
6    triggers : set Trigger,
7    conditions : set Condition,
8    actions : some Action
9  } // Trigger, Condition, and Action contain
10 // similar tuples
11 abstract sig Trigger {
12  devices : some Device,
13  attribute : one Attribute,
14  values : set Value
15 abstract sig Condition { ... }
16 abstract sig Action { ... }

```

---



---

### Listing 3 Excerpt of Environment model.

---

```

1  abstract sig Channel {
2  sensors : set Capability,
3  actuators : set Capability
4  one sig ch_temperature extends Channel {} {
5  sensors = cap_temperatureMeasurement
6  actuators = cap_switch + cap_thermostat + cap_ovenMode
7  one sig ch_luminance extends Channel {} {
8  sensors = cap_luminanceMeasurement
9  actuators = cap_switch + cap_switchLevel
10 one sig ch_motion extends Channel {} {
11  sensors = cap_motionSensor + cap_contactSensor
12  actuators = cap_switch

```

---

### 6.1 Smart Home Base Model

The overall smart home system is modeled as a set of Devices and a set of IoTApps, as shown in Listing 2. Each IoTApp contains its own set of Rules. Each Device has some associated state Attributes, each of which can assume one of a disjoint set of Values. Recall from Section 4, each rule contains its own set of Triggers, Conditions, and Actions. Each individual trigger, condition, and action is modeled as a tuple of one or more Devices, the relevant Attribute for that type of device, and one or more Values that are of interest to the trigger, condition, or action. Defined in Alloy, each of the listed entities is an abstract signature which is extended to a concrete model signature for each specific type of device, attribute, value, IoT app, behavioral rule, etc.

**Environment Modeling.** Apps can communicate both *virtually* within the cloud backend and *physically* via the devices they control. Virtual interactions fall into two main categories: (1) direct mappings, where one app triggers another by acting directly on a virtual device/variable watched by the triggered app; or (2) scheduling, where one rule calls—e.g., using the runIn API from SmartThings—to invoke a second rule after a delay. Physically mediated interactions occur indirectly via some physical *channel*, such as temperature. Our model—in contrast to others [20, 47]—directly supports detection of violations mediated via physical channels (cf. Listing 3). As part of our model of the overall SmartThings ecosystem, we include a mapping of each device to one or more physical Channels as either a sensor or an actuator.

### 6.2 Extracted IoT App Behavioral Rule Models

The second set of specifications required by the *Formal Analyzer* is the models automatically extracted from each individual IoT app. These specifications extend the base specifications described in Section 6.1 with specific relations for each individual IoT app.

**Listing 4** Excerpts from the generated specification for MultiSwitch (Listing 1)

```

1  one sig MultiSwitch extends IoTApp {
2    master : one Switch,
3    switches : some Switch }
4    { rules = r0 + r1 }
5
6  one sig r0 extends Rule {}{
7    triggers = r0_trig0
8    no conditions
9    actions = r0_act0 }
10 one sig r0_trig0 extends Trigger {}{
11   devices = MultiSwitch.master
12   attribute = Switch_State
13   value = ON }
14 one sig r0_act0 Action {}{
15   devices = MultiSwitch.switches
16   attribute = Switch_State
17   value = ON }
18
19 one sig r1 extends Rule {}{ ... }

```

Listing 4 partially shows the Alloy specification generated for the MultiSwitch app from Section 2. First, the new signature MultiSwitch extends the base IoTApp by adding fields for some Switch devices—a single master and multiple others—as well as constraining the inherited rules field to contain the two rules, r0 and r1, defined on Lines 6 and 19 as extensions of Rule. As described in Section 5, the *Behavioral Rule Extractor* generates the tuples for the triggers, conditions, and actions of each app’s rules from the behavioral rule graph. In this case, the entry point node corresponding to the switchOn method is translated into the r0\_trig0 signature (Line 10) while the action node of the BRG generates r0\_act0 (Line 14). The bundle of these specifications define all apps co-installed in the system.

### 6.3 Safety/Security Properties

In Section 3, we present seven classes of multi-app coordination threats that violate safety. In this section, we describe Alloy assertions drawn from the logical definitions of those threat classes. These assertions express the threats as *safety properties* that are expected to hold in the specifications extracted from each individual IoT app. We also draw upon prior work [20, 26, 47] to define specific unsafe or undesirable behaviors that may result from chain triggering. Table 1 summarizes A set of safety and security properties caused due to action-trigger coordination (T1), derived from the literature [20, 26, 47] and formally verified by IoTCom. In total, we consider 36 safety and security properties, 7 generic coordination threats introduced Section 3) and 29 properties listed in Table 1<sup>1</sup>.

As a concrete example of the seven coordination threats, the assertion T4 in Listing 5 corresponds to threat T4, presented in Section 3. In this snippet, we define the predicate are\_connected (Lines 7-11) which encodes the relation specified in Def. 3.3 for threat (T1). It relies on the two other predicates, match (Lines 2-3) and same\_channel (Lines 4-6), which correspond to the logical relations of the same names defined in Defs. 3.1 and 3.2 from Section 3. The fact on Line 13 converts are\_connected predicate to a field connected on the Rule signature. Line 17 then defines the assertion, which ensures that no rule is found in the transitive closure of it’s

<sup>1</sup>The complete specifications of these properties are available online at our project site [35].

**Table 1: A sample of safety and security properties caused due to *action-trigger coordination*, derived from the literature [20, 26, 47] and formally verified by IoTCom.**

Property	Description
P.1	DON’T turn on the AC WHEN mode is away
P.2	DON’T turn on the bedroom light WHEN door is closed
P.3	DON’T turn on dim light WHEN there is no motion
P.4	DON’T turn on living room light WHEN no one is home
P.5	DON’T turn on dim light WHEN no one is home
P.6	DON’T turn on light/heater WHEN light level changes
P.7	DON’T turn off heater WHEN temperature is low
P.8	DON’T unlock door WHEN mode is away
P.9	DON’T turn off living room light WHEN someone is home
P.10	DON’T turn off AC WHEN temperature is high
P.11	DON’T close valve WHEN smoke is detected
P.12	DON’T turn off living room light WHEN mode is away
P.13	DON’T turn off living room light WHEN mode is vacation
P.14	DO set mode to away WHEN no one is home
P.15	DO set mode to home WHEN someone is home
P.16	DON’T turn on heater WHEN mode is away
P.17	DON’T open door/window WHEN smoke is detected or mode is away
P.18	DON’T turn off security system WHEN no one is home
P.19	DON’T turn off the alarm WHEN smoke is detected
P.20	DON’T unlock the door WHEN light level changes
P.21	DON’T lock the door WHEN smoke is detected
P.22	DON’T open the door WHEN smoke is detected and heater is on
P.23	DON’T unlock the door WHEN smoke is detected and heater is on
P.24	DON’T open the door WHEN motion is detected and fan is on
P.25	DON’T unlock the door WHEN motion is detected and fan is on
P.26	DON’T open the door/window WHEN temperature changes
P.27	DON’T set mode WHEN temperature changes
P.28	DON’T set mode WHEN smoke is detected
P.29	DON’T set mode WHEN motion is detected and alarm is sounding

own connected rules, matching the definition of the threat class from Def. 3.7.

As a concrete example of a safety properties caused due to action-trigger coordination (T1), Listing 6 illustrates the Alloy assertion for one of the fine-grained safety properties analyzed by IoTCom (P.8 in Table 1). This assertion states that no rule (r) should have an action (a, Line 2) that results in a contact sensor (i.e., the door) being opened (Lines 4-5) while the home mode is Away (Lines 10-11).

**Listing 5** Example coordination threats (T1) and (T4) defined as assertions in Alloy. The are\_connected predicate and connect attribute encode the logical definition of T1.

```

1 // example relations defining coordination (cf. Section 3)
2 pred match[a : Action, t : Trigger] {
3   (some t.devices & a.devices) and (a.attribute = t.attribute)}
4 pred same_channel[a : Action, t : Trigger] {
5   (some c : Channel | (a.devices in c.actuators) and
6   (some t.devices & y.sensors))}
7 pred are_connected[r,r' : Rule] {
8   (some a : r.actions, t : r'.triggers {
9     (match[a, t] and (a.value in t.value)) or same_channel[a, t]}))
10 all a : r.actions, c : r'.conditions {
11   (match[a,c]) => (a.value in c.value) } )
12 // defines the 'connected' field so we can use transitive closure
13 fact { all r,r' : Rule | (r' in r.connected) <=> are_connected[r,r'] }
14 // action-trigger coordination
15 assert t1 { no Rule.connected }
16 // self coordination
17 assert t4 { no r : Rule | r in r.^connected }

```

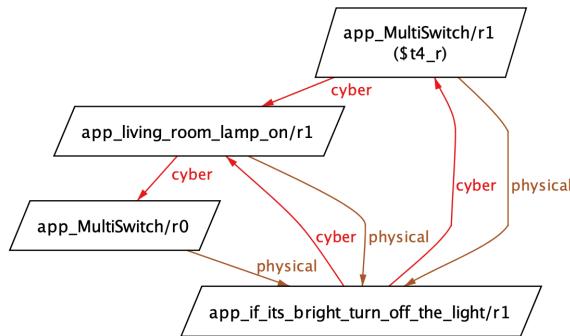
The property check is formulated as a problem of finding a valid trace that satisfies the specifications but violates the assertion. The returned solution encodes the sequence of rule activations leading to the violation. Given our running example (cf. Figure 1), the analyzer automatically detects a violation scenario, a visualization of which is shown in Figure 8, where the four rules form a cycle.

**Listing 6** Example Alloy assertion for the property DON'T unlock door WHEN location mode is Away.

```

1 assert P8 {
2   no r : IoTApp.rules, a : r.actions {
3     // DON'T open the door...
4     a.attribute = CONTACT_SENSOR_CONTACT_ATTR
5     a.values     = CONTACT_SENSOR_OPEN
6     // ... WHEN ...
7     ((some r' : r.*are_connected, t : r'.triggers {
8       (some r' : r.*are_connected, a' : r'.actions {
9         // ...mode is away
10        a'.attribute = MODE_ATTR
11        a'.values    = MODE_AWAY })) )}
```

IotCom's ability to detect violations in complex chains of interaction across both cyber and physical channels sets it apart from other research in the area.



**Figure 8:** An automatically detected violation scenario for our running example (cf. Fig. 1), where the four rules form a cycle.

## 7 EVALUATION

This section presents our experimental evaluation of IotCom, addressing the following research questions:

- **RQ1:** What is the overall accuracy of IotCom in identifying safety and security violations compared to other state-of-the-art techniques?
- **RQ2:** How well does IotCom perform in practice? Can it find safety and security violations in real-world apps?
- **RQ3:** What is the performance of IotCom's analysis realized atop static analysis and verification technologies?

**Experimental subjects.** Our experiments are all run on a multi-platform dataset of 3732 smart home apps drawn from two sources: (1) *SmartThings apps*: We gathered 404 SmartThings apps from the SmartThings public repository [51]. These apps are written in Groovy using the SmartThings Classic API platform. (2) *IFTTT applets*: We used the IFTTT dataset provided by Bastys et al. [14]. This dataset is in JSON format, with each object defining an IFTTT applet.

**Safety and Security Properties.** We use a set of 36 safety and security properties for all of our experiments, each encoded as an Alloy assertion as described in Section 6.3.

We performed the experiments on a MacBook Pro with a 2.2GHz 2-core Intel i7 processor and 16GB RAM, with the exception of the real-world app analysis in Section 7.2. Those experiments were

**Table 2: Safety violation detection performance comparison between SOTERIA, IoTSAN and IotCom.** True Positive (TP), False Positive (FP), and False Negative (FN) are denoted by symbols  $\checkmark$ ,  $\times$ ,  $\square$ , respectively. ( $X^{\#}$ ) represents the number # of detected instances for the corresponding symbol  $X$ .

Test Cases	SOTERIA*	IoTSAN	IotCom
<b>Individual Apps</b>			
ID1BrightenMyPath	$\checkmark$	$\checkmark$	$\checkmark$
ID2SecuritySystem	$\checkmark$	$\square^{\dagger}$	$\checkmark$
ID3TurnItOnOffandOnEvery30Secs	$\checkmark$	$\square$	$\checkmark$
ID4PowerAllowance	$\checkmark \square$	$(\square 2)$	$(\checkmark 2)$
ID5.1FakeAlarm	$\square$	$\square$	$\square$
ID6TurnOnSwitchNotHome	$\checkmark$	$\checkmark$	$\checkmark$
ID7ConflictTimeandPresenceSensor	$\checkmark$	$\square^{\ddagger}$	$\checkmark$
ID8LocationSubscribeFailure	$\checkmark$	$\checkmark$	$\checkmark$
ID9DisableVacationMode	$\times$	$\square$	$\checkmark$
<b>Bundles of Apps</b>			
Application Bundle 1	$\checkmark$	$\checkmark$	$\checkmark$
Application Bundle 2	$\checkmark$	$\square^{\dagger}$	$\checkmark$
Application Bundle 3	$\checkmark$	$\square^{\dagger}$	$\checkmark$
Application Bundle 4 <sup>#</sup>	$\square$	$\square^{\ddagger}$	$\checkmark$
Application Bundle 5 <sup>#</sup>	$\square$	$\square$	$\checkmark$
Application Bundle 6 <sup>#</sup>	$\square$	$\square$	$\checkmark$
Precision	90%	100%	<b>100%</b>
Recall	66.7%	25%	<b>93.8%</b>
F-measure	76.6%	40%	<b>96.8%</b>

\* results obtained from [21]

<sup>†</sup> IoTSAN did not generate the Promela model

<sup>‡</sup> SPIN crashing

<sup>#</sup> Benchmarks involving physical channels related violations.

performed as distributed jobs on a local cluster of 2 CPU/16 core Intel Xeon processors, with each job assigned up to 32GB RAM. We used Alloy 4.2 for model checking for all experiments.

### 7.1 Results for RQ1 (Accuracy)

To evaluate the effectiveness and accuracy of IotCom and compare it against other state-of-the-art techniques, we used the IoT-MAL [36] suite of benchmarks. This dataset contains custom SmartThings Classic apps, for which all violations, either singly or in groups, are known in advance—establishing a ground truth. As IotCom identifies safety/security violations arising from interactions of conflicting rules. Such rules can be defined within the scope of one app or multiple apps. Therefore, to perform a fair comparison with the state-of-the-art, we used benchmarks which incorporate violations in both individual apps and bundles of apps.

We faced two challenges while evaluating the accuracy of IotCom against the state-of-the-art: (1) Most analysis techniques—including HOMEGUARD [24], SOTERIA [20], and iRULER [57]—are not available; IoTSAN [47] was the lone exception. We also were not able to run IoTMON because only one component thereof is publicly accessible; its Groovy parser<sup>2</sup> is available, but the channels discovery and NLP components are not. SOTERIA [20] was evaluated using the IoT-MAL dataset, but the tool is not publicly available. Therefore, we rely on the results provided in the technical report [21]. (2) The

<sup>2</sup><https://github.com/nsslabcuus/IoTMon>

violations in the IoTMAL dataset do not involve physical channels. For evaluating this capability of the compared techniques, we developed three bundles, B4–B6, available online from the project website [35].

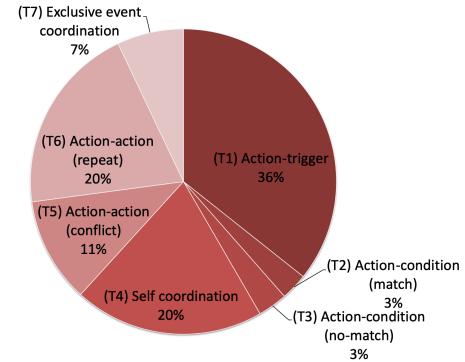
Table 2 summarizes the results of our experiments for evaluating the accuracy of IoTCom in detecting safety violations compared to the other state-of-the-art techniques. IoTCom succeeds in identifying all 9 known violations out of 10 in the individual apps, and all violations in 6 bundles of apps. Furthermore, IoTCom identifies two violations in the test case *ID4PowerAllowance*—namely, (T6) *repeated actions* and (T5) *conflicting actions*. Different from SOTERIA and IoTSAN, IoTCom captures schedule APIs; thus, it can identify the conflicting actions violation that was not detected by the other techniques. IoTCom misses only a single violation, in test case *ID5.1FakeAlarm*. This app generates a fake alarm using a smart device API not often used in SmartThings apps. Neither SOTERIA nor IoTSAN detected this violation.

IoTCom also successfully identifies potential safety and security violations arising from interactions between apps. Test bundles *B1 – B3* exhibit such violations using only virtual channels of interaction. Bundles *B4–B6* define violations due to *physical* interactions between apps. For example, B4 contains an interaction violation over the temperature channel that can result in the door being unlocked while the user is not present, violating one of the specific properties belonging to class (T1) *chain triggering*, while B5 and B6 contain unsafe behavior and infinite actuation loop, respectively. SOTERIA and IoTSAN cannot detect such violations that involve interactions over physical channel.

## 7.2 Results for RQ2 (Real-World Apps)

We further evaluated the capability of IoTCom to identify violations in real-world IoT apps. We randomly partitioned the subject systems of 3732 real-world SmartThings and IFTTT apps into 622 non-overlapping bundles, each comprising 6 apps, in keeping with the sizes of the bundles used in prior work [33, 47]. This partitioning simulates a realistic usage of IoT apps, where no restrictions prevent a user from installing any combination of IoT apps. The resulting bundles enabled us to perform several independent experiments. We evaluated each bundle against (1) the seven classes of interaction threats introduced in Section 3 and (2) a set of specific *action-trigger* coordinations adopted from the literature [20, 26, 47] (listed in Table 1), which we used to assess the capability of our approach to accommodate and detect scenario-based violations. More details on these properties, along with the specifications thereof are available from the project’s website [35]. Overall, IoTCom detected 2883 violations across the analyzed bundles of real-world IoT apps, with analysis failing on one of the 622 bundles.

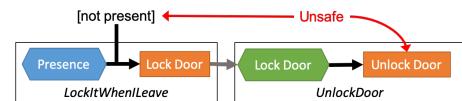
Figure 9 illustrates how the detected violations were distributed among the seven classes of multi-app coordination, presented in Section 3. Threats (T1) *Action-trigger*, (T4) *Self coordination*, and (T6) *Action-action (repeat)* were the most prevalent. These threats also appear in the motivating example from Section 2. The fact that action-trigger (T1) and action-action (T5, T6) coordination classes were the most frequently detected violations indicates that race conditions among actions would be the most likely consequence of multi-app coordination. Out of the 29 specific safety properties from



**Figure 9: Distribution of the detected violations across seven classes of multi-app coordination.**

class (T1) *Action-trigger*, IoTCom detects violations of 15 properties, where 72.3% (450 out of 621) of the bundles violate at least one property. In the following, we describe some of our findings.

**7.2.1 Violation of (T1) Action-trigger.** Figure 10 depicts a chain of virtual interactions that could lead to a door being left unlocked if misconfigured. The SmartThings app *LockItWhenLeave* locks the door when the user leaves the house, as detected by a presence sensor. The lock action triggers the IFTTT applet *Unlock Door*, which unlocks the door again. This violates one of our specific, scenario-based, action-trigger coordination properties.

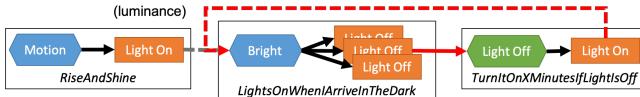


**Figure 10: Example violation of T1: Cyber coordination between apps may leave the door unlocked when no one is home. The first rule is guarded by a condition that the home owner not be present.**

This example also demonstrates IoTCom’s unique ability to consider logical conditions when evaluating interactions. The code of *LockItWhenLeave* does not specify a particular value for the presence sensor in the trigger for its rule; the entry method is invoked by any change to the presence sensor. Instead, the rule uses a condition to ensure it is only invoked when the user is *not* present. Other techniques, particularly those that require manual specification of the initial system configuration for analysis, may miss this violation by only considering the interaction when the user is present. IoTCom does not have such a limitation, and correctly identifies the violation.

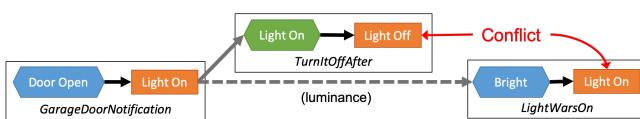
**7.2.2 Violation of (T4) Self Coordination via Physical Channel.** The chain of interactions shown in Figure 11 results in a loop that could continually turn a switch on and off, similar to our example from Section 2. This violation represents the self coordination threat (cf. T4). The loop involves three SmartThings apps: *RiseAndShine*, *TurnItOnXMinutesIfLightIsOff*, and *LightsOnWhenIArriveInTheDark*. *RiseAndShine* contains a rule activating some switch when motion

is detected. *LightsOnWhenIArriveInTheDark* controls a group of switches based on the light levels reported by light sensors. *TurnItOnXMinutesIfLightIsOff* switches a switch on for a user-specified period, then turns it back off.



**Figure 11: Example violation of T4: Lights continually turn off and on. Dashed line represents coordination via the *luminance* physical channel.**

When *RiseAndShine* activates its switch, it could trigger *LightsOnWhenIArriveInTheDark* via the *luminance* physical channel, switching all connected lights off. This event triggers *TurnItOnXMinutesIfLightIsOff*, which may re-enable one of the lights. This changes the luminance level, entering into an endless loop between *LightsOnWhenIArriveInTheDark* and *TurnItOnXMinutesIfLightIsOff*. IoTCom is uniquely capable of detecting this violation due to our support of physical channels, scheduling APIs, and arbitrarily long chains of interactions among apps.



**Figure 12: Example violation of T5: Both “on” and “off” commands sent to the same light due to the same event. Dashed line represents coordination via the *luminance* physical channel.**

**7.2.3 Violation of (T5) Action-action (conflict) via Physical Channel.** The three apps shown in Figure 12 lead to potentially unpredictable behavior due to competing commands to the same device, violating T5. They also interact in part over a physical channel that could not be detected by approaches that only consider virtual interaction between apps. The IFTTT applet *GarageDoorNotification* activates a switch when the garage door is opened. This triggers the action of SmartThings app *TurnItOffAfter*, which will turn off the light after a predefined period. At the same time, *GarageDoorNotification* may also have triggered the IFTTT applet *LightWarsOn* via a light sensor, interacting over the physical *luminance* channel. *LightWarsOn* would attempt to turn the light back on, producing an unpredictable result—a race condition—depending on which rule was executed first.

We also manually evaluated the accuracy of IoTCom on a sample of the real-world bundles. We first randomly selected 30 of the 622 bundles of six applications (approx. 5%), and acquired the source code for the apps in those bundles. We then manually examined the Groovy source or IFTTT definition of each app in each bundle to find violations of the seven classes of multi-app coordination presented in Section 3. The precision, recall, accuracy, and F-measure derived from that random bundles are summarized in Table 3.

**Table 3: Results obtained through a manual analysis of a sample of approx. 5% (30) of the 622 real-world bundles.**

Precision	Recall	Accuracy	F-measure
47.86%	94.92%	65.71%	62.64%

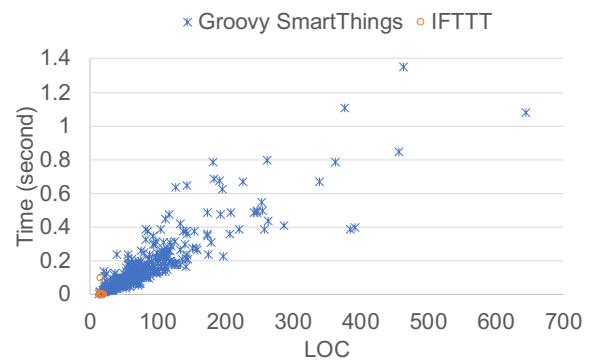
Note that the low precision (47.86%) is in part due to a quirk of IFTTT. IFTTT applets specify a “channel” to signify the device for the trigger and the action of each rule, identified by an ID value assigned by the platform and passed to the REST interface of the service invoked by the applet. The service controlling Android phones re-uses the same channel ID for multiple tasks; for example, detecting a connection to a particular WiFi network and changing the volume on the phone both correspond to the same channel ID. Our reference implementation of IoTCom interprets both of those to be the same device, so a rule that, for example, muted the phone when connecting to a particular network would be flagged as a violation of both T1 and T4, even though the actual action taken by the rule would not result in any coordination.

### 7.3 Results for RQ3 (Performance and Timing)

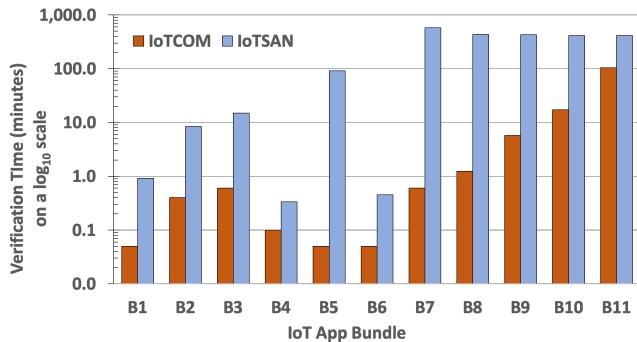
The last evaluation criteria are the performance benchmarks of model extraction and formal analysis of IoTCom on real-world apps drawn from the SmartThings and IFTTT repositories.

Figure 13 presents the time taken by IoTCom to extract rule models from the Groovy SmartThings apps and IFTTT applets. The scatter plot shows both the analysis time and the app size. According to the results, our approach statically analyzes and infer specifications for 98% of apps in less than one second.

We also measured the verification time required for detecting safety/security violations and compared the analysis time of IoTCom against that required by IoTSAN [47]. We checked all 36 safety and security properties against the app bundles. IoTSAN requires the initial configuration for each app in the bundle as part of the model to be analyzed. IoTCom, however, exhaustively examines all configurations that fall within the scope of the app model. To perform a fair comparison between the two approaches, we generated initial configurations for 11 bundles of apps and converted them into



**Figure 13: Scatter plot representing analysis time for behavioral rule extraction of IoT apps using IoTCom.**



**Figure 14: Comparing verification time by IoTCom and IoTSAN to perform the same safety violation detection across 11 bundles of real-world apps (in minutes, displayed on a log<sub>10</sub> scale). IoTCom remarkably outperforms IoTSAN (by 92.1% on average), without sacrificing the detection capability.**

a format supported by IoTSAN. We then ran the two techniques considering all valid initial configurations to avoid missing any violation.

Figure 14 depicts the total time taken by each approach to analyze *all* relevant configurations (rather than a single, user-selected configuration). Note that the analysis time is portrayed in a logarithmic scale. The experimental results show that the average analysis time taken by IoTCom and IoTSAN per bundle is 11.9 minutes (ranging from 0.05 to 104.78 minutes) and 216.9 minutes (ranging from 0.33 to 580.91 minutes), respectively. Overall, IoTCom remarkably outperforms IoTSAN in terms of the time required to analyze the same bundles of apps by 92.1% on average and by as much as 99.5%. The fact that IoTCom is able to effectively perform safety/security violation detection of real-world apps in just a few minutes (on an ordinary laptop), confirming that the presented technology is indeed feasible in practice for real-world usage.

#### 7.4 Threats to Validity

The major external threat to the validity of our results involves the study of a small set of benchmark programs developed and released by prior research work [20], so that we can directly compare our results with their previously reported results. To mitigate this threat and help determine whether our results may generalize, we conducted additional studies using 3732 real-world SmartThings and IFTTT apps. This enabled us to assess the capabilities of IoTCom in real-world scenarios and capture violations that have not been discovered by prior work. The primary threat to internal validity involves potential errors in the implementations of IoTCom and the infrastructure used to run IoTSAN. To overcome this threat, we extensively validated all of our tool components and scripts over the same baseline and configurations that have been used by prior work. The experimental data is also publicly available for external inspection. The main threat to construct validity relates to the fact that IoTCom identifies all potential safety violations that can occur but do not yet assess whether these violations can happen in reality (e.g., when the heater is switched on, it is difficult to identify the time that will be taken until the temperature

reaches a certain value because other factors can affect this process such as the size of the room).

## 8 DISCUSSION AND LIMITATIONS

For the sake of the feasibility demonstration of the presented ideas, this paper provides substantial supporting evidence for analyzing two of the most prominent IoT platforms, i.e., SmartThings home automation platform and IFTTT trigger-action platform. It would be interesting to see how IoTCom fares when applied to safety and security analysis of other IoT platforms, such as HealthSaaS [31], Microsoft Flow [44] and Zapier [63]. Given that we developed IoTCom based on the general trigger-condition-action paradigm, upon which these other IoT platforms are by and large relying, we believe the current analysis technique can be naturally extended to include such platforms. This forms a thrust of our future work.

Regarding the efforts required by end-users, IoTCom uses static analysis to automatically infer apps behavior, captured in BRG models. It then automatically translates such behavioral models into formal specifications in the Alloy language. The formal analysis part is also conducted automatically. However, the specifications for the base smart home Alloy model and the safety properties are manually developed once and can be reused by others. Thus, it poses a one-time cost to develop new properties to be analyzed. Each app specification is automatically extracted, independent of the other apps. So if a new app is added, the only thing the user needs to do is to run IoTCom’s behavioral rule extractor to automatically extract the specification for the new app without changing the other apps. Note that we also automatically identified device-specific attributes by parsing the documentation of SmartThings to extract capabilities and actions of devices.

Similar to any approach that relies on static analysis, IoTCom is subject to false positives. A fruitful avenue of future research is to strengthen IoTCom by incorporating dynamic analysis techniques. In principle, it should be possible to leverage dynamic analysis techniques to automatically confirm some of the violations, and potentially enforce the required safety policies, further lessening the manual analysis effort. Moreover, dynamic analysis can address dynamic features in Groovy apps, as they support reflection and can make HTTP requests at runtime [18].

## 9 RELATED WORK

IoT safety and security has received a lot of attention recently [2, 4, 6, 16, 17, 23, 25, 30, 40, 42, 46, 48, 52–56, 58–62, 65]. Here, we provide a discussion of the related efforts in light of our research. As depicted in Table 4, we compare IoTCom with the other existing approaches over various features provided by such analyzers. ContextIoT [38] analyzes individual IoT apps to prevent sensitive information leakage at run-time. However, it does not support the analysis of risky interactions among multiple apps. Soteria [20] and HOMEGUARD [24] are static analysis tools for detecting violations in multiple IoT apps. However, these techniques do not take into account physical channels, which can carry perilous interactions among apps, such as violations reported in section 7.2 as detected by IoTCom. Moreover, none of these techniques can handle the interactions between IoT apps and trigger-action platform services.

**Table 4: Comparing IoTCom with the state-of-the-art IoT analysis approaches.**

	ContextIoT [38]	SOTERIA [20]	HOMEGUARD [24]	IoTSAN [47]	IoTMON [26]	IoTCom
Physical channel analysis	X	X	X	X	✓	✓
Trigger-action applet analysis	X	X	X	✓	X	✓
Multi-app analysis	X	✓	✓	✓	✓	✓
Entire config. space analysis	X	X	X	X	X	✓

Along the same line, IoTSAN [47] detects violations in bundles of more than two apps. However, IoTSAN [47] first translates the Groovy code of the SmartThings apps to Java, limiting its analysis to just less than half of the devices supported by SmartThings [50]. In contrast, IoTCom directly analyzes Groovy code, supports large app bundles and all SmartThings device types, and is completely automated. IoTSAN [47], similar to Soteria [20] and HOMEGUARD [24], cannot detect violations mediated by physical channels.

IoTMON [26] is a purely static analysis technique that analyzes rules based solely on triggers, neglecting the conditions for specific actions. In contrast, IoTCom validates the safety of app interactions with more precision by effectively capturing logical conditions influencing the execution of app rules through a precise control flow analysis. Moreover, IoTMON, similar to many other techniques we studied, does not support the analysis of interactions between IoT apps and trigger-action platform services. To the best of our knowledge, IoTCom is the first IoT analysis technique for automated analysis of the entire configuration space, broadening the scope of the analysis beyond certain initial system configurations, required to specify in other existing techniques manually.

Other researchers have evaluated the security of IFTTT applets [3, 14, 27, 33]. Fernandes et al. [27] studied OAuth security in IFTTT, while Bastys et al. [14] used information flow analysis to highlight possible privacy, integrity, and availability threats. However, none of the studies examined the aforementioned IoT safety and security properties. In contrast, IoTCom performs large scale safety and security analysis, examining interactions between tens of IFTTT smart home applets. IoTCom also analyses bundles comprising both SmartThings Classic apps and IFTTT applets, demonstrating its unique cross-platform analytical capability.

## 10 CONCLUSION

This paper presents a novel approach for compositional analysis of IoT interaction threats. Our approach employs static analysis to automatically derive models that reflect behavior of IoT apps and interactions among them. The approach then leverages these models to detect safety and security violations due to interaction of multiple apps and their embodying physical environment that cannot be detected with prior techniques that concentrate on interactions

within the cyber boundary. We formalized the principal elements of our analysis in an analyzable specification language based on relational logic, and developed a prototype implementation, IoTCom, on top of our formal analysis framework. The experimental results of evaluating IoTCom against prominent IoT safety and security properties, in the context of thousands of real-world apps, corroborates its ability to effectively detect violations triggered through both virtual and physical interactions.

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by awards CCF-1755890 and CCF-1618132 from the National Science Foundation.

## REFERENCES

- [1] 2012. D. Jackson, *Software Abstractions*, 2nd ed. MIT Press, 2012. MIT Press.
- [2] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A Selcuk Uluagac. 2018. Peek-a-Boo: I see your smart home activities, even encrypted! *arXiv preprint arXiv:1808.02741* (2018).
- [3] Ioannis Agadakos, Chien-Ying Chen, Matteo Campanelli, Prashant Anantharaman, Monowar Hasan, Bogdan Copos, Tancrède Lepoint, Michael Locasto, Gabriela F. Ciocarlie, and Ulf Lindqvist. 2017. Jumping the Air Gap: Modeling Cyber-Physical Attack Paths in the Internet-of-Things. In *Proceedings of the 2017 Workshop on Cyber-Physical Systems Security and PrivaCy (CPS '17)*. ACM, New York, NY, USA, 37–48.
- [4] Bakr Ali and Ali Awad. 2018. Cyber and physical security vulnerability assessment for IoT-based smart homes. *Sensors* 18, 3 (2018), 817.
- [5] Apple Homekit 2018. Apple HomeKit. <https://www.apple.com/ios/home/>.
- [6] Noah Apthorpe, Dillon Reisman, and Nick Feamster. 2017. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *arXiv preprint arXiv:1705.06809* (2017).
- [7] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. 2018. A formal approach for detection of security flaws in the android permission system. *Formal Asp. Comput.* 30, 5 (2018), 525–544. <https://doi.org/10.1007/s00165-017-0445-z>
- [8] Hamid Bagheri, Alireza Sadeghi, Reyhaneh Jabbarvand Behrouz, and Sam Malek. 2016. Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 514–525. <https://doi.org/10.1109/DSN.2016.53>
- [9] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Trans. Software Eng.* 41, 9 (2015), 866–886. <https://doi.org/10.1109/TSE.2015.2419611>
- [10] Hamid Bagheri and Kevin J. Sullivan. 2013. Bottom-up model-driven development. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 1221–1224. <https://doi.org/10.1109/ICSE.2013.6606683>
- [11] Hamid Bagheri and Kevin J. Sullivan. 2016. Model-driven synthesis of formally precise, stylized software architectures. *Formal Asp. Comput.* 28, 3 (2016), 441–467. <https://doi.org/10.1007/s00165-016-0360-8>
- [12] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2014. TradeMaker: automated dynamic analysis of synthesized tradespaces. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 106–116. <https://doi.org/10.1145/2568225.2568291>
- [13] Hamid Bagheri, Chong Tang, and Kevin J. Sullivan. 2017. Automated Synthesis and Dynamic Analysis of Tradeoff Spaces for Object-Relational Mapping. *IEEE Trans. Software Eng.* 43, 2 (2017), 145–163. <https://doi.org/10.1109/TSE.2016.2587646>
- [14] Iuliu Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What?: Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1102–1119.
- [15] Jorge Blasco, Thomas M. Chen, Igor Muttik, and Markus Roggenbach. 2018. Detection of app collusion potential using logic programming. *J. Network and Computer Applications* 105 (2018), 88–104. <https://doi.org/10.1016/j.jnca.2017.12.008>
- [16] Christoph Busold, Stephan Heuser, Jon Rios, Ahmad-Reza Sadeghi, and N Asokan. 2015. Smart and secure cross-device apps for the internet of advanced things. In *International Conference on Financial Cryptography and Data Security*. Springer, 272–290.
- [17] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking

- in Commodity IoT. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1687–1704.
- [18] Z Berkay Celik, Earlene Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *arXiv preprint arXiv:1809.06962* (2018).
- [19] Z. Berkay Celik, Earlene Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. 2018. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *CoRR abs/1809.06962* (2018). arXiv:1809.06962 <http://arxiv.org/abs/1809.06962>
- [20] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 147–158.
- [21] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. Soteria: Automated IoT Safety and Security Analysis. *arXiv:cs.CR/1805.08876*
- [22] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
- [23] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Weng Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*.
- [24] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. 2018. Cross-App Interference Threats in Smart Homes: Categorization, Detection and Handling. *CoRR abs/1808.02125* (2018).
- [25] Chad Davidson, Tahsin Rezwana, and Mohammad A. Hoque. 2019. Smart Home Security Application Enabled by IoT: In *Smart Grid and Internet of Things*, Al-Sakib Khan Pathan, Zubair Md. Fadlullah, and Mohamed Guerroumi (Eds.). Springer International Publishing, Cham, 46–56.
- [26] Wenbo Ding and Hongxin Hu. 2018. On the Safety of IoT Device Physical Interaction Control. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 832–846.
- [27] Earlene Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action iot platforms. In *22nd Network and Distributed Security Symposium (NDSS 2018)*.
- [28] Garageio. 2019. IFTTT applet: If I arrive at my house then open my garage door. <https://ifttt.com/applets/213296p>.
- [29] Google home 2018. Google Home. <https://store.google.com/us/product/googlehome?hl=en-US>.
- [30] Arnoud Goudbeek, Kim-Kwang Raymond Choo, and Nhien-An Le-Khac. 2018. A Forensic Investigation Framework for Smart Home Environment. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018*, New York, NY, USA, August 1-3, 2018. IEEE, 1446–1451. <https://doi.org/10.1109/TrustComBigDataSE.2018.00201>
- [31] HEALTHSAAS 2020. HEALTHSAAS: THE INTERNET OF THINGS (IOT) PLATFORMFOR HEALTHCARE. [https://www.healthsaas.net/..](https://www.healthsaas.net/)
- [32] Households 2019. Households have 10 connected devices now, will rise to 50 by 2020, IT News, ET CIO. <https://cio.economictimes.indiatimes.com/news/internet-of-things/households-have-10-connected-devices-now-will-rise-to-50-by-2020/53765773>.
- [33] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. SafeChain: Securing Trigger-Action Programming From Attack Chains. *IEEE Trans. Information Forensics and Security* 14, 10 (2019), 2607–2622. <https://doi.org/10.1109/TIFS.2019.2899758>
- [34] IFTTT Documentation 2019. IFTTT Platform Documentation. <https://platform.ifttt.com/docs>.
- [35] IoTCom project website 2019. IoTCom project website. <https://sites.google.com/view/iotcom/home>.
- [36] iotmal 2019. IoTMAL benchmark app repository. <https://github.com/IoTBench/IoTBench-test-suite/tree/master/smartThings/smartThings-Soteria>.
- [37] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290.
- [38] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlene Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContextIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'17)*. San Diego, CA.
- [39] Sylvain Kubler, Kary Främling, and Andrea Buda. 2015. A standardized approach to deal with firewall and mobility policies in the IoT. *Pervasive and Mobile Computing* 20 (2015), 100 – 114. <https://doi.org/10.1016/j.pmcj.2014.09.005>
- [40] Brent Lagesse, Kevin Wu, Jaynie Shorb, and Zealous Zhu. 2018. Detecting Spies in IoT Systems using Cyber-Physical Correlation. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 185–190.
- [41] Li Li, Tegawende F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Oeteau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Inf. Softw. Technol.* 88 (2017), 67–95. <https://doi.org/10.1016/j.infsof.2017.04.001>
- [42] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically debugging IoT control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM, 133–142.
- [43] Niloofar Mansoor, Jonathan A. Saddler, Bruno Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04–09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 785–790. <https://doi.org/10.1145/3236024.3275534>
- [44] microsoftFlow 2020. microsoftFlow. <https://flow.microsoft.com>.
- [45] Eugene M. Myers. 1981. A Precise Inter-procedural Data Flow Algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 219–230.
- [46] Julie L. Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. 2017. IOTA: A Calculus for Internet of Things Automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 119–133.
- [47] Dang Tu Nguyen, Chengyu Song, Zhiyun Qian, Srikanth V. Krishnamurthy, Edward J. M. Colbert, and Patrick McDaniel. 2018. IoTSan: Fortifying the Safety of IoT Systems. In *Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 191–203.
- [48] Amir Rahmati, Earlene Fernandes, Kevin Eykholt, and Atul Prakash. 2018. Tyche: A risk-based permission model for smart homes. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 29–36.
- [49] SmartThings 2018. SmartThings Classic Documentation. <https://docs.smarthings.com/en/latest/ref-docs/reference.html>.
- [50] SmartThings capabilities reference 2018. SmartThings Classic capabilities reference. <https://docs.smarthings.com/en/latest/capabilities-reference.html>.
- [51] smarthings github 2018. SmartThings Community repository. <https://github.com/SmartThingsCommunity/SmartThingsPublic>.
- [52] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. 2017. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1501–1510.
- [53] A. Tekoglu and A. S. Tosun. 2016. A Testbed for Security and Privacy Analysis of IoT Devices. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. 343–348.
- [54] Hamishna Thapliyal, Nathan Ratajczak, Ole Wendorff, and Carson Labrado. 2018. Amazon Echo Enabled IoT Home Security System for Smart Home Environment. In *IEEE International Symposium on Smart Electronic Systems, iSES 2018 (Formerly iNiS)*, Hyderabad, India, December 17–19, 2018. IEEE, 31–36. <https://doi.org/10.1109/ISES.2018.00017>
- [55] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. Smartauth: User-centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Berkeley, CA, USA, 361–378.
- [56] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 3227–3231.
- [57] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11–15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1439–1453. <https://doi.org/10.1145/3319535.3345662>
- [58] Qi Wang, Wajih Ul Hassan, Adam M. Bates, and Carl A. Gunter. 2018. Fear and Logging in the Internet of Things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18–21, 2018*. [http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018\\_1A-2wang.pdf](http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_1A-2wang.pdf)
- [59] Judson Wilson, Riad S Wahby, Henry Corrigan-Gibbs, Dan Boneh, Philip Levis, and Keith Winstein. 2017. Trust but verify: Auditing the secure internet of things. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 464–474.
- [60] Fu Xiao, Le-Tian Sha, Zai-Ping Yuan, and Ru-Chuan Wang. 2017. VulHunter: A Discovery for unknown Bugs based on Analysis for known patches in Industry Internet of Things. *IEEE Transactions on Emerging Topics in Computing PP (09 2017)*, 1–1. <https://doi.org/10.1109/TETC.2017.2754103>
- [61] Muneeb Bani Yassein, Wail Mardini, and Ashwaq Khalil. 2016. Smart homes automation using Z-wave protocol. *2016 International Conference on Engineering*

- & MIS (ICEMIS) (2016), 1–6.
- [62] Tianlong Yu, Vyasa Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets-XIV)*. Article 5, 7 pages.
- [63] zapier 2020. zapier. <https://zapier.com/>.
- [64] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlito de Alvarenga. 2017. A survey of intrusion detection in Internet of Things. *J. Netw. Comput. Appl.* 84 (2017), 25–37. <https://doi.org/10.1016/j.jnca.2017.02.009>
- [65] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Hajo Jin Zhu. 2018. HoMonit: Monitoring Smart Home Apps from Encrypted Traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 1074–1088.

# DeepSQLi: Deep Semantic Learning for Testing SQL Injection

Muyang Liu  
University of Electronic Science and  
Technology of China  
Chengdu, China  
muyangl@foxmail.com

Ke Li\*  
University of Exeter  
Exeter, UK  
k.li@exeter.ac.uk

Tao Chen  
Loughborough University  
Loughborough, UK  
t.t.chen@lboro.ac.uk

## ABSTRACT

Security is unarguably the most serious concern for Web applications, to which SQL injection (SQLi) attack is one of the most devastating attacks. Automatically testing SQLi vulnerabilities is of ultimate importance, yet is unfortunately far from trivial to implement. This is because the existence of a huge, or potentially infinite, number of variants and semantic possibilities of SQL leading to SQLi attacks on various Web applications. In this paper, we propose a deep natural language processing based tool, dubbed DeepSQLi, to generate test cases for detecting SQLi vulnerabilities. Through adopting deep learning based neural language model and sequence of words prediction, DeepSQLi is equipped with the ability to learn the semantic knowledge embedded in SQLi attacks, allowing it to translate user inputs (or a test case) into a new test case, which is semantically related and potentially more sophisticated. Experiments are conducted to compare DeepSQLi with SQLmap, a state-of-the-art SQLi testing automation tool, on six real-world Web applications that are of different scales, characteristics and domains. Empirical results demonstrate the effectiveness and the remarkable superiority of DeepSQLi over SQLmap, such that more SQLi vulnerabilities can be identified by using a less number of test cases, whilst running much faster.

## CCS CONCEPTS

- Security and privacy → Web application security;
- Software and its engineering → Software testing and debugging.

## KEYWORDS

Web security, SQL injection, test case generation, natural language processing, deep learning

### ACM Reference Format:

Muyang Liu, Ke Li, and Tao Chen. 2020. DeepSQLi: Deep Semantic Learning for Testing SQL Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397375>

\*Li is the corresponding author of this paper. All authors made commensurate contributions to this paper. Li designed the research. Liu built the system and carried out experiments. Li and Chen supervised the research and interpreted experimental data and wrote the manuscript.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20...\$15.00

<https://doi.org/10.1145/3395363.3397375>

18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397375>

## 1 INTRODUCTION

Web applications have become increasingly ubiquitous and important since the ever prevalence of distributed computing paradigms, such as Cyber-Physical Systems and Internet-of-Things. Yet, they are unfortunately vulnerable to a variety of security threats, among which SQL injection (SQLi) has been widely recognised as one of the most devastating threats. Generally speaking, SQLi is an injection attack that embeds scripts in user inputs to execute malicious SQL statements over the relational database management system (RDBMS) running behind a Web application. As stated in the Akamai report<sup>1</sup>, SQLi attacks constituted 65.1% of the cyber-attacks on Web applications during November 2017 to March 2019. It also shows that the number of different types of Web attacks (e.g., XSS, LFI and PHPi) has ever increased, but none of them have been growing as fast as SQLi attacks. Therefore, detecting and preventing SQLi vulnerabilities are of ultimate importance to improve the reliability and trustworthiness of modern Web applications.

There are two common approaches to protect Web applications from SQLi attacks. The first one is customised negative validation, also known as input validation. Its basic idea is to protect Web applications from attacks by forbidden patterns or keywords manually crafted by software engineers. Unfortunately, it is difficult, if not impossible, to enumerate a comprehensive set of validation rules that is able to cover all types of attacks. The second approach is prepared statement that allow embedding user inputs as parameters, also known as placeholders. By doing so, attackers are difficult to embed SQLi code in user inputs since they are treated as value for the parameter. However, as discussed in [23] and [24], prepared statement is difficult to design given the sophistication of defensive coding guideline. In addition, there are many other terms, such as dynamic SQL of DDL statement (e.g., create, drop and alter) and table structure (e.g., names of columns, tables and schema) cannot be parameterised.

Test case generation, which build test suites for detecting errors of the system under test (SUT), is the most important and fundamental process of software testing. This is a widely used practice to detect SQLi vulnerabilities where test suites come up with a set of malicious user inputs that mimic various successful SQLi attacks, each of which forms a test case. However, enumerating a comprehensive set of semantically related test cases to fully test the SUT is extremely challenging, if not impossible. This is because there are a variety of SQLi attacks, many complex variants of which share similar SQL semantic. For example, the same attack can be

<sup>1</sup> <https://www.akamai.com/>

diversified by changing the encoding form, which appears to be different but is semantically equivalent, in order to evade detection.

Just like human natural language, malicious SQL statements have their unique semantic. Therefore, test case generation for detecting SQLi vulnerabilities can take great advantages by exploiting the knowledge from such semantic naturalness. For example, given a Web form with two user input fields, i.e., username and password, the following SQL statement conforms to a SQLi attack:

```
SELECT      *      FROM      members      WHERE
username='admin'+OR+'1'='1' AND password='__'
```

where the underlined parts are input by a user and constitute a test case that leads to an attack to the SUT. Given this SQLi attack, we are able interpret some semantic knowledge as follows.

- This is a tautology attack that is able to use any tautological clause, e.g., OR 1=1, to alter the statement in a semantically equivalent and syntactically correct manner without compromising its maliciousness.
- To meet the SQL syntax, an injection needs to have an appropriate placement of single quotation to complete a SQL statement. Therefore, the attack should be written as admin'+OR+'1'='1. In addition, the unnecessary part of the original statement can be commented by --.
- In practice, due to the use of some input filters like firewalls, blank characters will highly likely be trimmed by modern Web applications thus leading to the failure of admin' OR 1=1 to form a tautology attack. By replacing those blank characters with +, which is semantically equivalent, the attacker is able to disguise the attack in a more sophisticated manner.

Although semantic knowledge can be interpreted by software engineers, it is far from trivial to leverage such knowledge to automate the test case generation process.

Traditional test case generation techniques mainly rely on software engineers to specify rules to create a set of semantically tailored test cases, either in a manual [9, 19] or semi-automatic manner [3, 5, 31]. Such process is of limited flexibility due to the restriction of human crafted rules. Furthermore, it is expensive in practice or even be computationally infeasible for modern complex Web applications.

Recently, there has been a growing interest of applying machine learning algorithms to develop artificial intelligence (AI) tools that automate the test case generation process [11, 20, 27, 29] and [22]. This type of methods requires limited human intervention and do not assume any fixed set of SQL syntax. However, they are mainly implemented as a classifier that is used to diagnose whether a SQL statement (or part of it) is a valid statement or a malicious injection. To the best of our knowledge, none of those existing AI based tools are able to proactively generate semantically related SQLi attacks during the testing phase. There have been some attempts that take semantic knowledge into consideration. For example, [27] developed a classifier that considers the semantic abnormality of the OR phrase (e.g., OR 1=1 or OR 'i' in ('g', 'i')) in a tautology attack. Unfortunately, this method ignores other alternatives, which might be important when semantically generating SQLi attacks, to create tautology (e.g., we can use -- to comment out other code).

Bearing the above considerations in mind, this paper proposes a deep natural language processing (NLP) based tool<sup>2</sup>, dubbed DeepSQLi, which learns and exploits the semantic knowledge and naturalness of SQLi attacks, to automatically generate various semantically meaningful and maliciously effective test cases. Similar to the machine translation between dialects of the same language, DeepSQLi takes a set of normal user inputs or existing test case for a SQL statement (one dialect) and translates it into another test case (another dialect), which is semantically related but potentially more sophisticated, to form a new SQLi attack.

**Contributions.** The major contributions of this paper are:

- DeepSQLi is a fully automatic, end-to-end tool empowered by a tailored neural language model trained under the Transformer [32] architecture. To the best of our knowledge, this work is the first of its kind to adopt Transformer to solve problems in the context of software testing.
- To facilitate the semantic knowledge learning from SQL statement, five mutation operators are developed to help enrich the training dataset. Unlike the classic machine translation where only the sentence with the most probable semantic match would be of interest, in DeepSQLi, we extend the neural language model with Beam search [26], in order to generate more than one semantically related test case based on the given test case/normal inputs that needs translation. This helps to generate a much more diverse set of test cases, and thus providing larger chance to find more vulnerabilities.
- The effectiveness of DeepSQLi is validated on six real-world Web applications selected from various domains. They are with various scales and have a variety of characteristics. The results show that DeepSQLi is better than SQLMap, a state-of-the-art SQLi testing automation tool, in terms of the number of vulnerabilities detected and exploitation rate whilst running up to 6× faster.

**Novelty.** What make DeepSQLi *unique* are:

- It is able to translate any normal user inputs into some malicious inputs, which constitute to a test case. Further, it is capable of translating an existing test cases into another semantically related, but potentially more sophisticated test case to form a new SQLi attack.
- If the generated test case cannot achieve a successful SQLi attack, it would be fed back to the neural language model as an input. By doing so, DeepSQLi is continually adapted to create more sophisticated test cases, thus improving the chance to find previously unknown and deeply hidden vulnerabilities.

The remaining paper is organised as follows. Section 2 provides a pragmatic tutorial of neural language model used for SQLi in this paper. Section 3 delineates the implementation detail of our proposed DeepSQLi. Section 4 presents and discusses the empirical results on six real-world Web applications. Section 5 exploits the threats to validity and related works are overviewed in Section 6. Section 7 summarises the contributions of this paper and provides some thoughts on future directions.

<sup>2</sup> Source code and experimental data can be accessed at our project repository: <https://github.com/COLA-Laboratory/issta2020>.

## 2 DEEP NATURAL LANGUAGE PROCESSING FOR SQLI

In this section, we elaborate on the concepts and algorithms that underpin DeepSQLi and discuss how they were tailored to the problem of translating user inputs into test cases.

### 2.1 Neural Language Model for SQLi

Given a sequence of user inputs  $I_u = \{w_1, \dots, w_N\}$  where  $w_i$  is the  $i$ -th token, a language model aims to estimate a joint conditional probability of tokens of  $I_u$ . Since a direct calculation of this multi-dimensional probability is far from trivial, it is usually approximated by  $n$ -gram models [7] as:

$$\begin{aligned} P(w_1, \dots, w_N) &= \prod_{i=1}^N P(w_i | w_1, \dots, w_{i-1}) \\ &\approx \prod_{i=1}^N P(w_i | w_{i-n+1}, \dots, w_{i-1}) \end{aligned} \quad (1)$$

where  $N$  is the number of consecutive tokens. According to equation (1), we can see that the prediction made by the  $n$ -gram model is conditioned on its  $n - 1$  predecessor tokens. However, as discussed in [12],  $n$ -gram models are suffered from a sparsity issue where it does not work when predicting the first token in a sequence that has no predecessor.

To mitigate such issue, DeepSQLi makes use of neural language model as its fundamental building block. Generally speaking, it is a language model based on neural networks along with a probability distribution over sequences of tokens. In our context, such probability distribution indicates the likelihood to which a sequence of user inputs conform to a SQLi attack. For example, a sequence of inputs `admin' +OR+'1'='1` will have a higher probability since it is able to conform to a SQLi attack; whereas another sequence of inputs `OR SELECT AND 1`, which is semantically invalid from the injection point of view, will have a lower probability to become a SQLi attack.

Comparing to the  $n$ -gram model, which merely depends on the probability, the neural language model represents the tokens of an input sequence in a vectorised format, as known as word embedding which is an integral part in neural language model training. Empowered by deep neural networks, a neural language model is more flexibility with predecessor tokens having longer distances thus is resilient to data sparsity.

In DeepSQLi, we adopt a neural language model for token-level sequence modeling, given that a token is the most basic element in SQL syntax. In other words, given a sequence of user inputs  $I_u$ , the neural language model aims to estimate the joint probability of the inclusive vectorised tokens.

### 2.2 Multi-head Self-Attention in Neural Language Model

Attention mechanisms, which allow modelling of dependencies without regarding to their distance in sequences, have recently become an integral part of sequence generation tasks [25]. Among them, self-attention is an attention mechanism that has the ability to represent relationship between tokens in a sequence. For example,

we can better understand the token "`1`" in the sequence "`OR 2 > 1`" by answering three questions: "*why to compare* (i.e., *what kind of attack*)", "*how to compare*" and "*who to compare with*". Self-attention have been successfully applied to deal with various NLP tasks, such as machine translation [32], speech recognition [10] and music generation [17]. In DeepSQLi, the self-attention is calculated by the scaled dot-product attention proposed by Vaswani et al [32]:

$$\begin{aligned} \mathbf{Q} &= \mathbf{XW}_Q, \mathbf{K} = \mathbf{XW}_K, \mathbf{V} = \mathbf{XW}_V \\ A(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V} \end{aligned} \quad (2)$$

where  $\mathbf{X}$  is the word embedding, i.e., the vector representation, of the input sequence,  $\mathbf{Q}$  is a matrix consists of a set of packed queries,  $\mathbf{K}$  and  $\mathbf{V}$  are keys and values matrices whilst  $d$  is the dimension of the key. In particular,  $\mathbf{Q}$ ,  $\mathbf{K}$  and  $\mathbf{V}$  are obtained by multiplying  $\mathbf{X}$  by three weight matrices  $\mathbf{W}_Q$ ,  $\mathbf{W}_K$  and  $\mathbf{W}_V$ .

In order to learn more diverse representations, we apply a multi-head self-attention in DeepSQLi given that it has the ability to obtain more information from the input sequence by concatenating multiple independent self-attentions. Specifically, a multi-head self-attention can be formulated as:

$$MA(\mathbf{Q}_X, \mathbf{K}_X, \mathbf{V}_X) = [A_1(\mathbf{Q}_1, \mathbf{K}_1, \mathbf{V}_1) \otimes \dots \otimes A_h(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)] \cdot \mathbf{W}_a, \quad (3)$$

where  $\otimes$  is a concatenation operation,  $\mathbf{W}_a$  is a weight matrix and  $h$  is the length of parallel attention layers, also known as heads. Since each head is a unique linear transformation of the input sequence representation as queries, keys and values, the concatenation of multiple independent heads enables the information extraction from different subspaces thus leading to more diverse representations.

### 2.3 Encoder-Decoder (Seq2Seq) Model

To train the neural language model, we adopt Seq2Seq—a general framework consists of an encoder and a decoder—in DeepSQLi. In particular, Transformer [32] is used to build the Seq2Seq model in DeepSQLi instead of those traditional recurrent neural network (RNN) [33] and convolutional neural network (CNN) [18], given its state-of-the-art performance reported in many Seq2Seq tasks.

Figure 1 shows an illustrative flowchart of the Seq2Seq model in DeepSQLi. The encoder takes a sequence of vector representations of  $N$  tokens, denoted as  $\mathbf{X} = \{x_1, \dots, x_N\}$ , which embed semantic information of tokens; whilst the input of the decoder is another sequence of vector representations of  $\bar{N}$  tokens, denoted as  $\mathbf{Y} = \{y_1, \dots, y_{\bar{N}}\}$ . Note that  $N$  is not necessary to be equal to  $\bar{N}$ . The purpose of this Seq2Seq model is to learn a conditional probability distribution over the output sequence conditioned on the input sequence, denoted as  $P(y_1, \dots, y_{\bar{N}} | x_1, \dots, x_N)$ . As for the example shown in Figure 1 where the input sequence is `OR 2 > 1` and the output sequence is `|| True`, the conditional probability is  $P(|| \text{True} | \text{OR } 2 > 1)$ .

More specifically, the encoder in the left hand side of Figure 1 consists of  $N$  identical layers, each of which has a multi-head self-attention mechanism sub-layer and a deep feed-forward neural

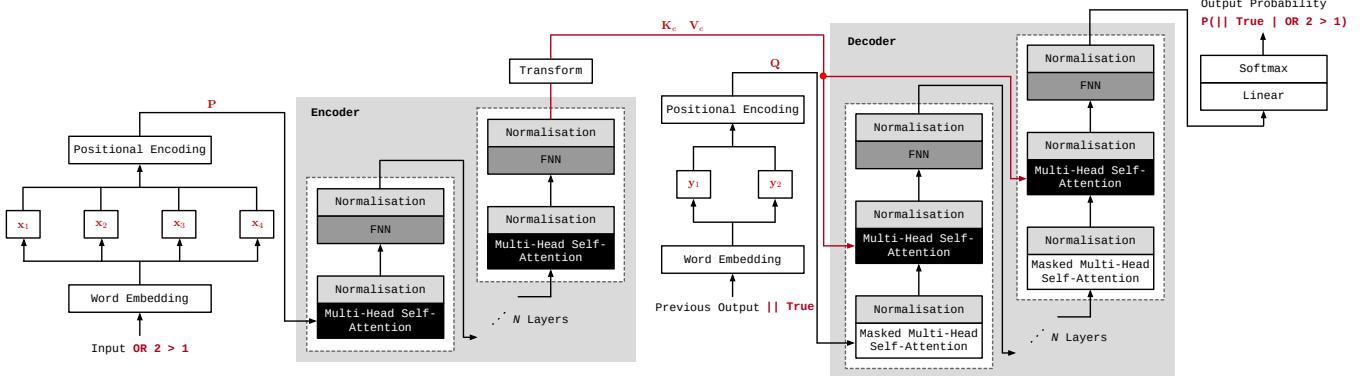


Figure 1: An illustrative working flowchart of the encoder-decoder (Seq2Seq) model in DeepSQLi.

network (FFN) sub-layer. The first sub-layer is the multi-head self-attention. As shown in Figure 1, the output of the multi-head self-attention, calculated by equation (3), is denoted as  $Z_1$ . It is supplemented with a residual connection  $\epsilon_{Z_1}$  to come up with the output  $N_1$  after a layer-normalisation. This process can be formulated as:

$$N_1 = \text{layer-normalisation}(Z_1 + \epsilon_{Z_1}). \quad (4)$$

Afterwards,  $N_1$  is fed to the second sub-layer, i.e., a FNN, to carry out a non-linear transformation. Specifically, the basic mechanism of the FNN is formulated as:

$$z_2 = \text{FFN}(n_1) = \max(0, n_1 W_1 + b_1) W_2 + b_2, \quad (5)$$

where  $n_1$  is a vector of  $N_1$ . Thereafter, the output of the FNN, i.e.,  $Z_2$ , will be transformed to two matrices  $K_e$  and  $V_e$  after being normalized to  $N_2$ . It is worth noting that  $K_e$  and  $V_e$  are the output of the encoder whilst they embed all information of the input sequence "OR 2 > 1".

As for the decoder shown in the right hand side of Figure 1, it takes  $K_e$  and  $V_e$  output from the encoder as a part of inputs for predicting a sequence of semantically translated vector representation  $Y$ . The decoder is also composed of a stack of  $N$  identical layers, each of which consists of a masked multi-head self-attention, a multi-head self-attention and a FNN sub-layers. In particular, the computational process of the multi-head self-attention and the FNN sub-layers is similar to that of the encoder, except that  $K_e$  and  $V_e$  are used as the  $K$  and  $V$  of equation (2) in the multi-head self-attention sub-layer. As for the masked multi-head self-attention sub-layer, it is used to avoid looking into tokens after the one under prediction. For example, the multi-head self-attention masks the second token "True" when predicting the first one "||".

In principle, the Transformer used to do Seq2Seq allows for significantly more parallel processing and has been reported as a new state-of-the-art. Unlike the RNN, which takes tokens in a sequential manner, the multi-head attention computes the output of each token independently, without considering the order of words. Since the SQLi inputs used in DeepSQLi are sequences with determined semantics and syntax, it may lead to a less effective modelling of the sequence information without taking any order of tokens into consideration. To take such information into account, the Transformer supplements each input embedding with a vector

called positional encoding (PE). Specifically, PE is calculated by sine and cosine functions with various frequencies:

$$\text{PE}_i = \begin{cases} \{\sin(\frac{i}{10000^{\frac{2}{d_e}}}), \dots, \sin(\frac{i}{10000^{\frac{2d_e}{d_e}}})\}, & \text{if } i \% 2 == 0 \\ \{\cos(\frac{i}{10000^{\frac{2}{d_e}}}), \dots, \cos(\frac{i}{10000^{\frac{2d_e}{d_e}}})\}, & \text{if } i \% 2 == 1 \end{cases}, \quad (6)$$

where  $d_e$  is the dimension of the vector representation,  $i$  represents the index of the token in the sequence.  $\text{PE}_i$  indicates that the sine variable is added to the even position of the token vector whilst the cosine variable is added to the odd position. Thereafter, the output token vector  $x_i$  is updated by supplementing  $\text{PE}_i$ , i.e.,  $p_i = x_i + \text{PE}_i$ . By doing so, the relative position between different embedding can be inferred without demanding costs .

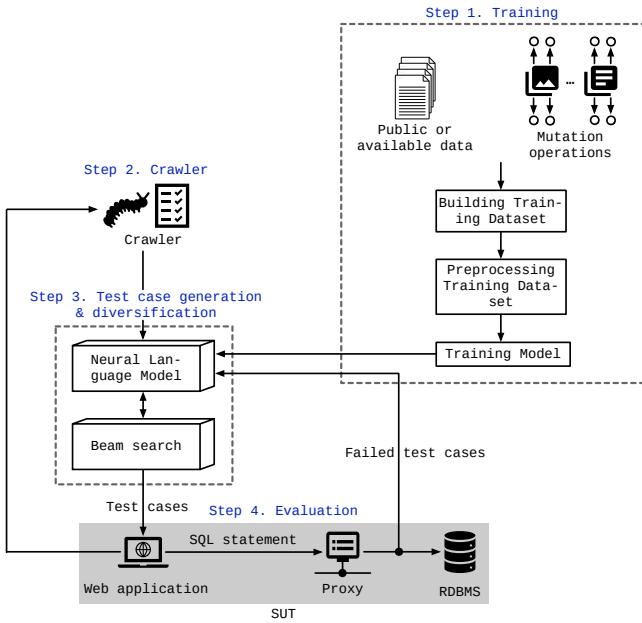
### 3 END-TO-END TESTING WITH DEEPSQLI

DeepSQLi is designed as an end-to-end tool, covering all stages in the penetration testing [13]. As shown in Figure 2, the main workflow of DeepSQLi consists of four steps: *training*, *crawler*, *test case generation & diversification* and *evaluation*, each of which is outlined as follows:

**Step 1: Training:** This is the first step of DeepSQLi where a neural language model is trained by the Transformer. Agnostic to the Web application, the training dataset can either be summarised from historical testing repository or, as what we have done in this work, mined from publicly available SQLi test cases/data. The collected test cases/data would then be paired, mutated and preprocessed to constitute the training dataset. We will discuss this in detail in Section 3.1.

**Step 2: Crawler:** Once the model is trained, DeepSQLi uses a crawler (e.g., the crawler of the Burp Suite project<sup>3</sup>) to automatically parse the Web links of the SUT (as provided by software engineers). The major purpose of the crawler is to extract the fields for user inputs in the HTML elements, e.g., `<input>` and `<textarea>`, which are regarded as the injection points for a SQLi attack. These injection points, along with their default values, serve as the starting point for the neural language model to generate SQLi test cases.

<sup>3</sup> <https://portswigger.net/burp>



**Figure 2: The architecture and workflow of DeepSQLi.**

**Step 3: Test Case Generation & Diversification:** The neural language model of DeepSQLi is able to generate tokens with different probabilities leading to a test case. To fully exploit such advantage for exploring a diverse set of alternative test cases, in the translation phase, we let the neural language model to generate and explore tokens with the top  $m$  probability instead of merely using only the highest one. This is achieved by Beam search [26], a heuristic that guides the neural language model to generate  $m$  test cases based on the ranked probabilities. This will be elaborated in detail in Section 3.2.

**Step 4: Evaluation:** Based on the test cases generated at Step 3, we randomly choose one and feed it into the SUT for evaluation. In particular, to avoid compromising the SUT, it is equipped with a proxy, i.e., SQL Parser<sup>4</sup>, before the RDBMS to identify whether or not a malicious SQL statement can achieve a successful attack. To improve the chance of detecting different vulnerabilities, DeepSQLi stops exploring a specific vulnerability once it has been found through a test case.

It is worth noting that DeepSQLi does not discard unsuccessful test cases which fail to achieve attacks as identified by the proxy. Instead, they are fed back to the neural language model to serve as a starting point for a re-translation (i.e., go through Step 3 again). This comes up with a closed-loop until the test case successfully injects the SUT or the maximum number of attempts is reached. By this means, DeepSQLi grants the ability to generate not only the standard SQLi attacks, but also those more sophisticated ones which would otherwise be difficult to create.

### 3.1 Training of Neural Language Model

DeepSQLi is agnostic to the SUT since we train a neural language model to learn the semantic knowledge of SQLi attack that is independent to an actual Web application. Therefore, DeepSQLi, once being trained sufficiently, can be applied to a wide range of SUT as the semantic knowledge of SQLi is easily generalisable. The overall training procedure is illustrated in Figure 2. In the following subsections, we further elaborate some key procedures in detail.

**3.1.1 Building Training Dataset.** In practice, it is possible that the SUT has accumulated a good amount of test cases from previous testing runs, which can serve as the training dataset. Otherwise, since we are only interested to learn the SQL semantics for injections, the neural language model of DeepSQLi can be trained with any publicly available test cases for SQLi testing regardless to the Web applications, as what we have done in this paper.

Since our purpose is to translate and generate a semantically related test case based on either a normal user input or another test case, the test cases in the training dataset, which work on the same injection point(s), need to appear in input-output pairs for training. In particular, an input-output pair  $(A, B)$  is valid if any of the following conditions is met:

- $A$  is a known normal user input and  $B$  is a test case. For example, `https://xxx/id=7` can be paired with `https://xxx/id=7 union select database()`.
- $A$  is a test case whilst  $B$  is another one, which is different but still conform to the same type of attack. For example,  $A$  is `' OR 1=1 --` and  $B$  is `' OR 5<7 --`. It is clear that both of them lead to a semantically related tautology attack.
- $A$  is a test case whilst  $B$  is an extended one based on  $A$ , thereby we can expect that  $B$  is more sophisticated but in a different attack type. For example,  $A$ : `' OR 1=1; --` can be paired with  $B$ : `' or 1=1; select group_concat(schema_name) from information_schema. schema" ta; --`, which belongs to a different type of attack, i.e., a piggybacked queries attack extended from  $A$ .

In this work, we manually create input-output test case pairs to build the training dataset based on publicly available SQLi test cases, such as those from public repositories, according to the aforementioned three conditions. More specifically, the training dataset is built according to the following two steps.

**Step 1:** We mined the repositories of fuzzing test or brute force tools from various GitHub projects<sup>5</sup>, given that they often host a large amount of test case data in their library and these data are usually arranged according to the types of attacks (along with normal user inputs). This makes us easier to constitute input-output pairs according to the aforementioned conditions. In particular, it is inevitable to devote non-trivial manual efforts to classifying and arranging some more cluttered data.

**Step 2:** When analyzing the mined dataset, we found it is difficult to constitute input-output pairs for the disguise attack. For example, a test case containing `' OR 1=1 --` may fail to

<sup>4</sup> <http://www.sqlparser.com>.

<sup>5</sup> <https://tinyurl.com/wh94b8t>

**Table 1: Description of five mutation operators used in DeepSQLi to enrich the training dataset.**

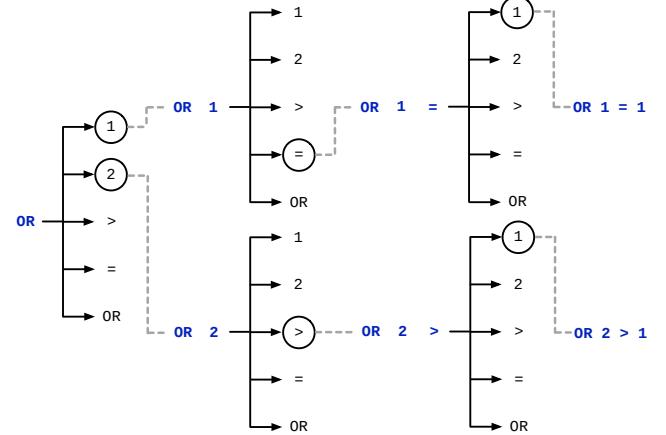
Operators	Explanation	Input	Example Output
Predicate	Mutation by using relational predicates without changing the expression's logical results. In particular, the relational predicates are $\{<, >, \leq, \geq, between, in, like\}$ .	and 8>= 56 and'1'in ('m', 'y')	
Unicode	Mutation from a character to its equivalent Unicode format.	#	%23
ASCII	Mutation from a character to its equivalent ASCII encoding format.	a	CHAR(97)
Keywords	Confusion of the capital and small letters of keywords in a test case.	select	seLeCt
Blank	Replace the blank character in a test case with an equivalent symbol.	or 1	or/**/1

inject the SUT due to the existence of the standard input validation. Whereas a successful SQLi attack can be formulated by simply change '`OR 1=1 --`' to '`%27%20OR%201=1%20--`', which is semantically the same but in a different encoding format. This is caused by the rare existence of semantically similar SQLi attacks based on manipulating synonyms and encoding formats from those public repositories. To tackle this issue, five mutation operators, as shown in Table 1, are developed to further exploit the test cases obtained from Step 1. By doing so, we can expect a semantically similar test case that finally conforms to a disguise attack. In principle, these mutation operators are able to enrich the test case samples in the training dataset.

**3.1.2 Preprocessing Training Dataset.** After building the training dataset, we then need to preprocess the data samples by generalisation and word segmentation to eliminate unnecessary noise in the training data. Notably, unlike classic machine learning approaches [4] that generalise all the words in a data sample, we only generalise the user inputs, the table name and column name to unify tokens "`[normal]`", "`[table]`" and "`[column]`". This is because other words and characters, including text-, encoding-, blank characters-, quotes-transforms, are specifically tailored in a SQLi attack, thereby they should not be generalised. For example, considering a test case "`admin'%20or%203<7;--`" in the training dataset, it is converted into a sequence as "`['[normal]', ' ', '%20', 'or', '%2', '3', '<', '7', ';', '--']`" after the generalisation.

**3.1.3 Training the Model.** In DeepSQLi, the neural language model is trained under the Transformer architecture. As suggested by Vaswani et al. [32], a stochastic optimization algorithm called Adam [21], with the recommended settings of  $\beta_1 = 0.9$ ,  $\beta_2 = 0.98$  and  $\epsilon = 10^{-9}$ , is used as the training algorithm.

To prevent overfitting, a 10-fold cross validation is applied in the training process with Adam to optimize the setting of some hyperparameters, including the number of layers in the encoder and the decoder, the number of hidden layers and neurons in FNN, as well as the number of heads used in the self-attention mechanism of the Transformer. In particular, the following loss function is used in



To make the generated test cases be more diversified, Beam search [26] is used to extend and guide the neural language model to generate a set of semantically related test cases. In a nutshell, instead of only focusing on the most probable tokens, Beam search selects the  $m$  most probable tokens at each time step given all previously selected tokens, where  $m$  is the beam width. Afterwards, by leveraging the neural language model, it merely carries on the search from these  $m$  tokens and discard the others. Figure 3 shows an example when  $m = 2$  and the corpus size is 5. According to the first token "`OR`", 2 sequences `OR 1` and `OR 2` with the highest probability are selected from 5 candidate sequences at the first time step. Then, the 2 sequences with the highest probability from the 10 possible output sequences are selected at each subsequent step until the last token in the sequence is predicted. From the above search process, we can also see that Beam search does not only improve the diversity, but also amplify the exploration of search space thus improve the accuracy. In DeepSQLi, we set the beam width as 5, which means that the neural language model creates 5 different test cases for each input. For example, considering the case when the input is '`OR 1=1`', then DeepSQLi could generate the following outputs: '`OR 5<7`', '`|| 5<7`', '`+OR+1=1`', '`OR 1=1`', and '`OR 1=1--`'.

In principle, this diversification procedure enables DeepSQLi to find more vulnerabilities since the output test cases translated from a given test case (or normal user inputs) are diversified.

## 4 EVALUATION

In this section, the effectiveness of DeepSQLi is evaluated by comparing with SQLmap<sup>6</sup>, the state-of-the-art SQLi testing automation tool [28], on six real-world Web applications. Note that SQLmap was not designed with automated crawling, thus it is not directly comparable with DeepSQLi under our experimental setup. To make a fair comparison and to mitigate the potential bias, we extend SQLmap with a crawler, i.e., the Burp Suite project used in DeepSQLi.

Our empirical study aims to address the following three research questions (RQs):

- **RQ1: Is DeepSQLi effective for detecting SQLi vulnerabilities?**
- **RQ2: Can DeepSQLi outperform SQLmap for detecting SQLi vulnerabilities?**
- **RQ3: How does DeepSQLi perform on SUT with advanced input validation in contrast to SQLmap?**

All experiments were carried out on a desktop with Intel i7-8700 3.20GHz CPU, 32GB memory and 64bit Ubuntu 18.04.2.

### 4.1 Experiment Setup

**4.1.1 Subject SUT.** Our experiments were conducted on six SUT<sup>7</sup> written in Java and with MySQL as the back-end database system. All these SUT are real-world commercial Web applications used by many researchers in this literature, e.g., [16]. In particular, there are two levels of input validation equipped with these SUT:

- **Essential:** This level filters the most commonly used keywords in SQLi attacks, e.g., '`AND`' and '`OR`'.

<sup>6</sup> <http://SQLmap.org/>.

<sup>7</sup>More detailed information can be found at <http://examples.codecharge.com>.

**Table 2: Real-world SUT used in our experiments.**

SUT	LOC	Servlets*	DBIs	KV
<i>Employee</i>	5,658	7 (10)	23	25
<i>Classifieds</i>	10,949	6 (14)	34	18
<i>Portal</i>	16,453	3 (28)	67	39
<i>Office Talk</i>	4,543	7 (64)	40	14
<i>Events</i>	7,575	7 (13)	31	26
<i>Checkers</i>	5,421	18 (61)	5	44

\* The # of accessible Servlets (the total # of Servlets).

- **Advanced:** This is an enhanced level<sup>8</sup> that additionally filters some special characters, which are rarely used but can still be part of a SQLi attack, e.g., '`&&`' and '`||`'.

Table 2 provides a briefing of the SUT considered in our experiments. In particular, these SUT cover a wide spectrum of Web applications under real-world settings. They are chosen from various application domains with different scales in terms of the line-of-code (LOC) and they involve various database interactions (DBIs)<sup>9</sup>. Furthermore, the number of Servlets and the number of known SQLi vulnerabilities (dubbed as KV in Table 2) are set the same as the existing study [16]. It is worth noting that both the number of accessible Servlets and their total amount are shown in Table 2 since not all Servlets are directly accessible.

To mitigate any potentially biased conclusion drawn from the stochastic algorithm, each experiment is repeated 20 times for both DeepSQLi and SQLmap under every SUT.

**4.1.2 Training Dataset.** In our experiments, the dataset used to train DeepSQLi is constituted by SQLi test cases, consisting of a diverse type of SQLi attacks, collected from various projects in GitHub. We make the training dataset publicly accessible to ensure that our results are reproducible<sup>10</sup>. Afterwards, we preprocess the training dataset by pairing and mutation according to the steps and conditions discussed in Section 3.1.1. In particular, the number of paired SQLi test case instances is 19,220, all of which can be directly used for training DeepSQLi. After using the mutation operators, the training dataset is significantly diversified and the number of training data instances is increased to 56,841.

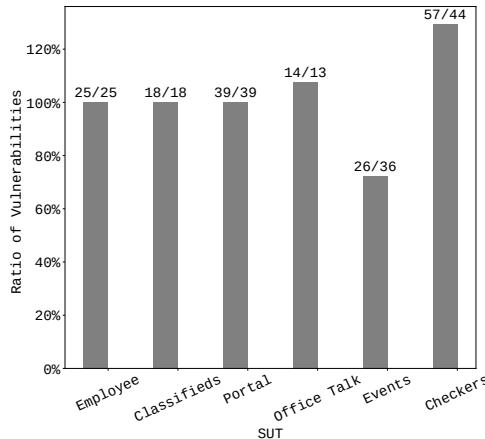
**4.1.3 Quality Metrics.** The following three quality metrics are used in our empirical evaluations.

- **Number of vulnerabilities found:** We use the number of SQLi vulnerabilities identified by either DeepSQLi or SQLmap as a criterion to evaluate their ability for improving the security of the underlying SUT.
- **Number of test cases and exploitation rate (ER):** In order to evaluate the ability for utilising computational resources, we keep a record of the total number of test cases generated by either DeepSQLi or SQLmap, denoted as  $T_{\text{total}}$ .

<sup>8</sup>This level is usually switched off, because more advanced security mechanism can often make the performance of Web application worse off.

<sup>9</sup>The number of SQL statements that can access user inputs.

<sup>10</sup> <https://tinyurl.com/wh94b8t>



**Figure 4: Ratio of # of vulnerabilities identified by DeepSQLi to that of SQLmap.**

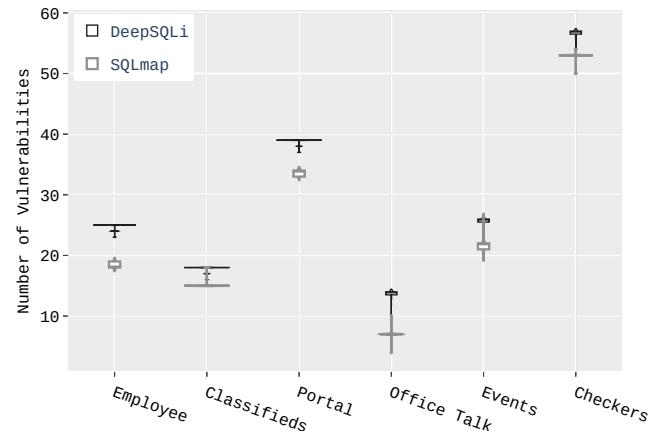
In addition, we also chase the number of test cases that successfully lead to SQLi attacks, denoted as  $T_{\text{success}}$ . Thereafter, ER is the ratio of  $T_{\text{success}}$  to  $T_{\text{total}}$ , i.e.,  $\frac{T_{\text{success}}}{T_{\text{total}}}$ .

- **CPU wall clock time:** In order to evaluate the computational cost required by either DeepSQLi or SQLmap, we keep a record of the CPU wall clock time used by them for testing the underlying SUT.

## 4.2 The Effectiveness of DeepSQLi

In this section, we firstly examine DeepSQLi on SUT with the *essential* input validation. Figure 4 presents the ratio of the average number of vulnerabilities identified by DeepSQLi (over 20 runs) to that of known vulnerabilities. From this result, we can clearly see that DeepSQLi is able to identify all known SQLi vulnerabilities for 5 out of 6 SUT, except the *Events*. This might be caused by the crawler which fails to capture all injection points in *Events*. In contrast, it is worth noting that DeepSQLi is able to identify more SQLi vulnerabilities than those reported in [14] for *Office Talk* and *Checker*. This is a remarkable result that demonstrates the ability of DeepSQLi for identifying previously unknown and deeply hidden vulnerabilities of the underlying black-box SUT.

To further understand why DeepSQLi is effective for revealing the SQLi vulnerabilities, Table 3 shows the injectable SQL statements and the related test cases generated in our experiments. Specifically, in Example 1, DeepSQLi is able to learn that the input test case, which is an unsuccessful SQLi attack, has failed due to a missing quotation mark. Thereby, it generates another semantically related test case which did find a vulnerability. In Example 2, we see more semantically sophisticated amendments though exploiting the learned semantic knowledge of SQL: the initially failed test case `and 3=4` is translated into another semantically related one, i.e., `and' '9`, which failed to achieve an attack again. Subsequently, in the next round, DeepSQLi then translates it into a more sophisticated and successful test case `and%208=9`, which eventually leads to the discovery of a vulnerability. Likewise, for Examples 3 to 5,



**Figure 5: Violin charts of the number of SQLi vulnerabilities identified by DeepSQLi (black lines) and SQLmap (gray lines) on six SUT with *essential* input validation across 20 runs.**

the input test cases have been translated into another semantically related and more sophisticated test cases.

**Answer to RQ1: Because of the semantic knowledge learned from previous SQLi attacks, DeepSQLi has shown its effectiveness in detecting SQLi vulnerabilities. It is worth noting that DeepSQLi is able to uncover more vulnerabilities that are deeply hidden and previously unknown in the SUT.**

## 4.3 Performance Comparison Between DeepSQLi and SQLmap

Under the SUT with the *essential* input validation, Table 4 shows the comparison results of the total number of test cases generated by DeepSQLi and SQLmap (dubbed as #total) versus the amount of test cases leading to successful attacks (dubbed as #success). From these results, it is clear that DeepSQLi is able to fully test the SUT with fewer test cases than SQLmap. In addition, as demonstrated by the better ER values achieved by DeepSQLi, we can conclude that DeepSQLi is able to better utilise test resources.

To have an in-depth analysis, Figure 5 uses violin charts to visualise the distribution of the number of SQLi vulnerabilities identified by DeepSQLi and SQLmap on all six SUT across 20 runs. From this comparison result, it is clear that DeepSQLi is able to find more vulnerabilities than SQLmap at all instances. In particular, as shown in Figure 5, the violin charts of DeepSQLi experienced much less variance than that of SQLmap. This observation implies that it is capable of producing more robust results by learning and leveraging the semantic knowledge embedded in the previous SQLi test cases.

Table 5 shows the comparison results of the CPU wall clock time required for running DeepSQLi and SQLmap. From this comparison result, we find that DeepSQLi runs much faster than SQLmap. In particular, it achieves up to 6× faster running time at the SUT *Portal*. By cross referencing with the results shown in Table 4, we can see that SQLmap generates much more test cases than DeepSQLi. It is worth noting that more test cases do not indicate any better

**Table 3: Examples of inputs and outputs of DeepSQLi**

Example 1	<i>Input</i>	SELECT card_type_id FROM card_types WHERE card_type_name=' and+1=(select count(*) from )-- ';
	<i>Output</i>	SELECT card_type_id FROM card_types WHERE card_type_name=' 'and+1=(select count(*) from ) -- ';
Example 2	<i>Input</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= and 3=4 ;
	<i>Output</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= and ''7 ;
	<i>Input</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= and ''9 ;
	<i>Output</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= and%208=9 ;
Example 3	<i>Input</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= 2;
	<i>Output</i>	SELECT category_id, name, par_category_id FROM categories WHERE category_id= 2;delete from members;
Example 4	<i>Input</i>	SELECT id, level FROM members WHERE member_login ='%20  %20'h ='h AND member_password='# ';
	<i>Output</i>	SELECT id, level FROM members WHERE member_login ='%200r%20'h ='h AND member_password='-- ';
Example 5	<i>Input</i>	INSERT INTO members (member_login,member_password,name,email,location,work_phone,home_phone) VALUES ("select database()", "test", "test", "test", "test", "1", "1");
	<i>Output</i>	INSERT INTO members (member_login,member_password,name,email,location,work_phone,home_phone) VALUES ("sElEct* database()", "test", "test", "test", "test", "1", "1");

**Table 4: Comparison results of the # of total/successful test cases generated by DeepSQLi and SQLmap, and the ER values.**

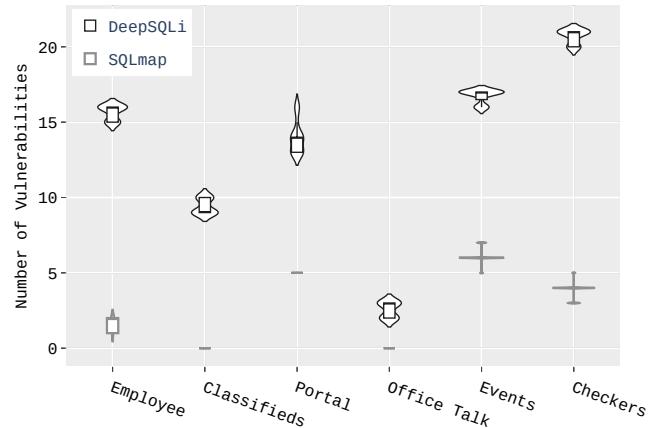
SUT	DeepSQLi		SQLmap	
	#total/#success	ER	#total/#success	ER
<i>Employee</i>	5563/473	8.50%	34851/1534	4.40%
<i>Classifieds</i>	4512/340	7.54%	25954/1046	4.03%
<i>Portal</i>	8657/740	8.55%	63001/2244	3.56%
<i>Office Talk</i>	2998/260	8.67%	9451/462	4.89%
<i>Events</i>	5331/487	9.14%	32136/1440	4.48%
<i>Checkers</i>	13463/1077	8.00%	67919/3352	4.94%

**Table 5: Comparison of the CPU wall clock time (in second) used to run DeepSQLi and SQLmap over all 20 runs.**

SUT	DeepSQLi	SQLmap
<i>Employee</i>	355	1177
<i>Classifieds</i>	236	931
<i>Portal</i>	357	2105
<i>Office Talk</i>	166	384
<i>Events</i>	259	1094
<i>Checkers</i>	519	2228

contribution for revealing SQLi vulnerabilities, because those test cases might be either unsuccessful or redundant as reflected by the lower ER values achieved by SQLmap. In addition, generating a much higher number of (useless) test cases makes SQLmap much slower than DeepSQLi.

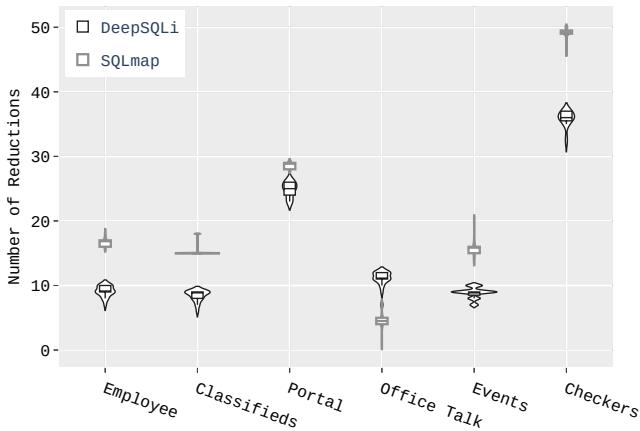
**Answer to RQ2: DeepSQLi is able to find significantly more SQLi vulnerabilities than SQLmap, with a better utilization of the testing resource as evidenced by the better exploitation rates. DeepSQLi also runs much faster than SQLmap on up to 6× better.**

**Figure 6: Violin charts of the number of SQLi vulnerabilities identified by DeepSQLi (black lines) and SQLmap (gray lines) on SUT with advanced input validation across 20 runs.**

#### 4.4 Performance Comparison Between DeepSQLi and SQLmap on SUT with Advanced Input Validation

The previous subsections have validated the effectiveness and performance of DeepSQLi on SUT with *essential* input validation. In this subsection, we switch on the *advanced* input validation in the SUT, aiming to assess the performance of DeepSQLi against SQLmap under more complicated and challenging scenarios.

Figure 6 presents the comparison results for the number of SQLi vulnerabilities found by DeepSQLi and SQLmap on SUT under *advanced* input validation. As can be seen, DeepSQLi finds remarkably more SQLi vulnerabilities than SQLmap. It is worth noting that there are a few runs where SQLmap failed to detect any vulnerability at *Employee*, *Classifieds* and *Office Talk*. In contrast, DeepSQLi shows consistently better performance for finding considerably more SQLi vulnerabilities.



**Figure 7: Violin charts of the number of reductions of SQLi vulnerabilities identified by DeepSQLi (black lines) and SQLmap (gray lines) on SUT with *advanced* input validation across 20 runs.**

In order to evaluate the sensitivity of DeepSQLi and SQLmap to the strength of input validation, we compare the number of vulnerabilities found on SUT under *advanced* input validation with that on SUT under *essential* input validation. By cross referencing Figure 4, we can observe some reductions on the number of vulnerabilities identified by both tools, as shown in Figure 7. However, in contrast to SQLmap, it is clear that DeepSQLi is much less affected by the strengthened input validation in 5 out of 6 SUT, demonstrating its superior capability of revealing SQLi vulnerabilities in more complicated scenarios. This better result achieved by DeepSQLi can be attributed to the effective exploitation of the semantic knowledge learned from previous SQLi test cases.

**Answer to RQ3:** Under the advanced input validation, DeepSQLi leads to much better results than that of SQLmap, which can hardly find any vulnerability at all in a considerable number of runs. In general, DeepSQLi is much less affected by the Web applications with strengthened input validation.

## 5 THREATS TO VALIDITY

As with any empirical study, the biases from the experiments can affect the conclusion drawn. As a result, we study and conclude this work with the following threats to validity in mind.

The metrics and evaluation method used is a typical example of the construct threats, which concern whether the metrics/evaluation method can reflect what we intend to measure. In our work, the metrics studied are widely used in this field of research [6, 30] and they serve as quality indicator for different aspects of SQLi testing. To mitigate the randomness introduced by the training, we repeat 20 experiment runs for each tool under a SUT. To thoroughly report our results without losing information, the distributions about the number of SQLi vulnerability found, which is the most important metric, have also been plotted in violin charts.

Internal threats are concerned with the degree of control on the studies, particularly related to the settings of the deep learning algorithm. In our work, the hyperparameters of Transformer are automatically tuned by using Adam and 10-fold cross validation, which is part of the training. The internal parameters of Adam itself were configured according to the suggestions from Vaswani et al. [32]. The crawler and proxy DeepSQLi are also selected based on their popularity, usefulness and simplicity.

External threats can be linked to the generalisation of our findings. To mitigate such, firstly, we compare DeepSQLi with a state-of-the-art tool, i.e., SQLmap. This is because SQLmap is the most cost-effective tool and has been widely used as a baseline benchmark [1, 3, 30]. Secondly, we study six real-world SUT that are widely used as standard benchmarks for SQLi testing research [14–16]. Despite that all the SUT are based on Java, they come with different scales, number of vulnerabilities and the characteristics, thus they are representatives of a wide spectrum of Web applications. In future work, we aim to evaluate DeepSQLi on other Web applications developed in different programming languages.

## 6 RELATED WORK

In order to detect and prevent SQLi attacks, different approaches have been proposed over the last decade, including anomalous SQL query matching, static analysis, dynamic analysis, etc.

Several approaches aim to parse or generate SQLi statements based on specific SQL syntax. For example, Halfond and Orso [14] proposed AMNESIA, a combination of dynamic and static analysis approach. At the static analysis stage, models of the legitimate queries that applications can generate is automatically built. In the dynamic analysis phase, AMNESIA uses runtime monitoring to check whether dynamically generated queries match the model. Mao et al. [8] presented an intention-oriented detection approach that converted SQL statement into a deterministic finite automaton and detect SQL statement to determine if the statement contains an attack. These aforementioned approaches, unlike DeepSQLi, heavily rely on fixed syntax or source code to estimate unknown attacks. In addition, they are not capable of learning the semantic knowledge from the SQL syntax.

Among other tools, BIOFUZZ[30] and  $\mu$ 4SQLi[2] are black-box and automated testing tools that bear some similarities to DeepSQLi. However, they have not made the working source code publicly available or the accessible code is severely out-of-date, thus we cannot compare them with DeepSQLi in our experiments. Instead, here we qualitatively compare them in light with the contributions of DeepSQLi:

- BIOFUZZ [30] is a search-based tool that generates test cases using context-free grammars based fitness function. However, as the fitness function is artificially designed based solely on prior knowledge, it is difficult to fully capture all possible semantic knowledge of SQLi attacks. This is what we seek to overcome with DeepSQLi. Further, the fact that BIOFUZZ relies on fixed grammars may also restrict its ability to generate semantically sophisticated test cases.
- $\mu$ 4SQLi [2] is an automated testing tool that uses mutation operators to modify the test cases, with a hope of finding

more SQLi vulnerabilities. However, these mutation operators are designed with a set of fixed patterns, thus it is difficult to generate new attacks that have not been captured in the patterns. In DeepSQLi, we also design a few mutation operators, but they are solely used to enrich the training data, which would then be learned by the neural language model. In this way, DeepSQLi is able to create attacks that have not been captured by patterns in the training samples.

SQLMap is used as state-of-the-art in the experiments because it is a popular and actively maintained penetration SQLi testing tool, which has been extensively used in both academia [1, 3, 30] and industry [28]. Here we also make a qualitative comparison of differences between SQLMap and DeepSQLi.

- SQLMap relies on predefined syntax to generate test cases. Such practice, as discussed in the paper, cannot actively learn and search for new SQLi attacks, as the effectiveness entirely depends on the manually crafted rules, which may involve errors or negligence. On the other hand, DeepSQLi learns the semantics from SQL statements and test cases. Such a self-learning process allows it to generalize to previously unforeseen forms of attacks. Our experiments have revealed the superiority of DeepSQLi in detecting the SQLi vulnerabilities.
- SQLMap generates new test cases from scratch. DeepSQLi, in contrast, allows intermediately unsuccessful, yet more malicious test cases to be reused as the inputs to generate new one. This enables it to build more sophisticated test cases incrementally and is also one of the reasons that leads to a faster process of DeepSQLi over SQLMap.

Recently, the combination of machine learning and injection-based vulnerability prevention has become popular [4][27][27][3][29]. Among others, Kim et al. [20] used internal query trees from the log to train a SVM to classify whether an input is malicious. Sheykhkhanloo et al. [27] trained a neural network with vectors which assigned for attacks to classify SQLi attacks. Appelt et al. [3] presented ML-Driven, an approach generates test cases with context-free grammars and train a random forest to detect SQLi vulnerability as the software runs. Jaroslaw et al. [29] applied neural networks to detect SQLi attacks. Their purpose is to build a model that learns the normal input and predicts whether the next input of user is malicious or not.

Unlike DeepSQLi, none of the work aims to generate SQLi test cases for conducting end-to-end testing on the Web application. Moreover, DeepSQLi leverages deep NLP to explicitly learn the semantic knowledge of the SQL for generating the whole sequence of SQLi test case. In particular, DeepSQLi translates the normal user inputs into test cases, which, when fail, would then be re-entered into DeepSQLi to generate more sophisticated SQLi attacks.

## 7 CONCLUSION AND FUTURE WORK

SQLi attack is one of the most devastating cyber-attacks, the detection of which is of high importance. This paper proposes DeepSQLi, a SQLi vulnerability detection tool that leverages on the semantic knowledge of SQL to generate test cases. In particular, DeepSQLi relies on the Transformer to build a neural language model under the Seq2Seq framework, which can be trained to explicitly learn the

semantic knowledge of SQL statements. By comparing DeepSQLi with SQLmap on six real-world SUT, the results demonstrate the effectiveness of DeepSQLi and its remarkable improvement over the state-of-the-art tool, whilst still being effective on Web applications with *advanced* input validation.

In future work, we will extend DeepSQLi with incrementally updated neural language model by using the generated test cases as the testing runs. Moreover, expanding DeepSQLi to handle other vulnerabilities, e.g., Cross-site Scripting, is also within our ongoing research agenda.

## ACKNOWLEDGMENT

Li was supported by UKRI Future Leaders Fellowship (Grant No. MR/S017062/1).

## REFERENCES

- [1] Dennis Appelt, Nadia Alshahwan, and Lionel C. Briand. 2013. Assessing the Impact of Firewalls and Database Proxies on SQL Injection Testing. In *FITTEST'13: Proc. Workshop of the 2013 Future Internet Testing - First International*. 32–47.
- [2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *ISSTA'14: Proc. of the 2014 International Symposium on Software Testing and Analysis*. 259–269.
- [3] Dennis Appelt, Cu D. Nguyen, Annibale Panichella, and Lionel C. Briand. 2018. A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls. *IEEE Trans. Reliability* 67, 3 (2018), 733–757.
- [4] Davide Ariu, Igino Corona, Roberto Tronci, and Giorgio Giacinto. 2015. Machine Learning in Security Applications. *Trans. MLDI* 8, 1 (2015), 3–39.
- [5] Ilies Benikhlef, Chenghong Wang, and Sangirov Gulomjon. 2016. Mutation based SQL injection test cases generation for the web based application vulnerability testing. In *ICENCE'16: Proc. of the 2nd International Conference on Electronics, Network and Computer Engineering*.
- [6] Josip Bozic, Bernhard Garn, Dimitris E. Simos, and Franz Wotawa. 2015. Evaluation of the IPO-Family algorithms for test case generation in web security testing. In *ICST'15 Workshops: Proc. Workshop of the 2015 Eighth IEEE International Conference on Software Testing, Verification and Validation*. 1–10.
- [7] Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, Jennifer C. Lai, and Robert L. Mercer. 1992. An Estimate of an Upper Bound for the Entropy of English. *Computational Linguistics* 18, 1 (1992), 31–40.
- [8] Chenyu, Mao, Fan, and Guo. 2016. Defending SQL Injection Attacks based-on Intention-Oriented Detection. In *ICCS'16: Proc. of the 11th International Conference on Computer Science & Education*. IEEE, 939–944.
- [9] Mark Curphey and Rudolph Arawo. 2006. Web application security assessment tools. *IEEE Security & Privacy* 4, 4 (2006), 32–41.
- [10] Linhao Dong, Shuang Xu, and Ba Xu. [n.d.]. Speech-Transformer: A No-Recurrence Sequence-to-Sequence Model for Speech Recognition. In *ICASSP'18: Proc. of the 2018 IEEE International Conference on Acoustics, Speech and Signal Processing*.
- [11] Rohan Doshi, Noah Apthorpe, and Nick Feamster. 2018. Machine Learning DDoS Detection for Consumer Internet of Things Devices. In *SP Workshop'18: Proc. of the 2018 IEEE Security and Privacy*. 29–35.
- [12] David Guthrie, Ben Allison, Wei Liu, Louise Guthrie, and Yorick Wilks. 2006. A Closer Look at Skip-gram Modelling. In *LREC'06: Proc. of the 5th International Conference on Language Resources and Evaluation*. 1222–1225.
- [13] Halfond, William GJ, Choudhary, Shauvik Roy, Orso, and Alessandro. 2009. Penetration testing with improved input vector identification. In *ICST'09: Proc. of the 2nd International Conference on Software Testing Verification and Validation*. 346–355.
- [14] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *ASE'05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 174–183.
- [15] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *SIGSOFT'06: Proc. of the 14th ACM International Symposium on Foundations of Software Engineering*. 175–185.
- [16] William G. J. Halfond, Alessandro Orso, and Pete Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Trans. Software Eng.* 34, 1 (2008), 65–81.
- [17] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Ian Simon, Curtis Hawthorne, Noam Shazeer, Andrew M. Dai, Matthew D. Hoffman, Monica Dinulescu, and Douglas Eck. 2019. Music Transformer: Generating Music with

- Long-Term Structure. In *ICLR'19: Proc. of the 7th International Conference on Learning Representations*.
- [18] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A Convolutional Neural Network for Modelling Sentences. In *ACL'14: Proc. of the 52nd Association for Computational Linguistics*. 655–665.
- [19] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. In *ICSE'09: Proc. of the 31st International Conference on Software Engineering*. 199–209.
- [20] Mi-Yeon Kim and Dong Hoon Lee. 2014. Data-mining based SQL injection attack detection using internal query trees. *Expert Syst. Appl.* 41, 11 (2014), 5416–5430.
- [21] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR'15: Proc. of the 52nd Association for Computational Linguistics*.
- [22] Huichen Li, Xiaojun Xu, Chang Liu, Teng Ren, Kun Wu, Xuezhi Cao, Weinan Zhang, Yong Yu, and Dawn Song. 2018. A Machine Learning Approach to Prevent Malicious Calls over Telephony Networks. In *SP'18: Proc. of the 2018 IEEE Symposium on Security and Privacy*. 53–69.
- [23] Ofer Maoz and Amichai Shulman. 2004. SQL injection signatures evasion. *Imperva, Inc., Apr* (2004).
- [24] Stuart McDonald. 2002. SQL Injection: Modes of attack, defense, and why it matters. *White paper, GovernmentSecurity.org* (2002).
- [25] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent Models of Visual Attention. In *NIPS'14: Proc. of the 2014 Neural Information Processing Systems*. 2204–2212.
- [26] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI'14: Proc. of the 2014 Programming Language Design and Implementation*. 419–428.
- [27] Naghmeh Moradpoor Sheykhanloo. 2017. A Learning-based Neural Network Model for the Detection and Classification of SQL Injection Attacks. *IJCWT* 7, 2 (2017), 16–41.
- [28] Sanjib Sinha. 2018. SQL Mapping. In *Beginning Ethical Hacking with Kali Linux*. Springer, 221–258.
- [29] Jaroslaw Skaruz and Franciszek Seredyński. 2007. Recurrent neural networks towards detection of SQL attacks. In *IPDPS'07: Proc. of the 21th International Parallel and Distributed Processing Symposium*. 1–8.
- [30] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based security testing of web applications. In *SBST'14: Proc. of the 7th International Workshop on Search-Based Software Testing*. 5–14.
- [31] Wei Tian, Jufeng Yang, Jing Xu, and Guannan Si. 2012. Attack Model Based Penetration Test for SQL Injection Vulnerability. In *COMPSAC'12: Proc. Workshops of the 36th Annual IEEE Computer Software and Applications*. 589–594.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS'17: Proc. of the 2017 Neural Information Processing Systems*. 5998–6008.
- [33] Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. 2015. Grammar as a Foreign Language. In *NIPS'15: Proc. of the 2015 Neural Information Processing Systems*. 2773–2781.

# Dependent-Test-Aware Regression Testing Techniques

Wing Lam

University of Illinois

Urbana, Illinois, USA

winglam2@illinois.edu

August Shi

University of Illinois

Urbana, Illinois, USA

awshi2@illinois.edu

Reed Oei

University of Illinois

Urbana, Illinois, USA

reedoei2@illinois.edu

Sai Zhang\*

Google

Kirkland, Washington, USA

saizhang@google.com

Michael D. Ernst

University of Washington

Seattle, Washington, USA

mernst@cs.washington.edu

Tao Xie

Peking University

Beijing, China

taoxie@pku.edu.cn

## ABSTRACT

Developers typically rely on regression testing techniques to ensure that their changes do not break existing functionality. Unfortunately, these techniques suffer from flaky tests, which can both pass and fail when run multiple times on the same version of code and tests. One prominent type of flaky tests is order-dependent (OD) tests, which are tests that pass when run in one order but fail when run in another order. Although OD tests may cause flaky-test failures, OD tests can help developers run their tests faster by allowing them to share resources. We propose to make regression testing techniques dependent-test-aware to reduce flaky-test failures.

To understand the necessity of dependent-test-aware regression testing techniques, we conduct the first study on the impact of OD tests on three regression testing techniques: test prioritization, test selection, and test parallelization. In particular, we implement 4 test prioritization, 6 test selection, and 2 test parallelization algorithms, and we evaluate them on 11 Java modules with OD tests. When we run the orders produced by the traditional, dependent-test-unaware regression testing algorithms, 82% of human-written test suites and 100% of automatically-generated test suites with OD tests have at least one flaky-test failure.

We develop a general approach for enhancing regression testing algorithms to make them dependent-test-aware, and apply our approach to 12 algorithms. Compared to traditional, unenhanced regression testing algorithms, the enhanced algorithms use provided test dependencies to produce orders with different permutations or extra tests. Our evaluation shows that, in comparison to the orders produced by unenhanced algorithms, the orders produced by enhanced algorithms (1) have overall 80% fewer flaky-test failures due to OD tests, and (2) may add extra tests but run only 1% slower on average. Our results suggest that enhancing regression testing algorithms to be dependent-test-aware can substantially reduce flaky-test failures with only a minor slowdown to run the tests.

\*Most of Sai Zhang's work was done when he was at the University of Washington.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397364>

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

flaky test, regression testing, order-dependent test

### ACM Reference Format:

Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397364>

## 1 INTRODUCTION

Developers rely on *regression testing*, the practice of running tests after every change, to check that the changes do not break existing functionalities. Researchers have proposed a variety of regression testing techniques to improve regression testing. These regression testing techniques produce an order (a permutation of a subset of tests in the test suite) in which to run tests. Examples of such traditional regression testing techniques include test prioritization (run all tests in a different order with the goal of finding failures sooner) [37, 40, 45, 56, 57, 62], test selection (run only a subset of tests whose outcome can change due to the code changes) [20, 32, 35, 51, 52, 70], and test parallelization (schedule tests to run across multiple machines) [38, 41, 50, 63].

Unfortunately, regression testing techniques suffer from *flaky tests*, which are tests that can both pass and fail when run multiple times on the same version of code and tests [42, 47–49]. Flaky-test failures mislead developers into thinking that their changes break existing functionalities, wasting the developers' productivity as they search for non-existent faults in their changes. In fact, Herzig et al. [33] reported that test result inspections, which include verifying whether a test failure is a flaky-test failure, can cost about \$7.2 million per year for products such as Microsoft Dynamics. Flaky-test failures can also lead developers to start ignoring test failures during builds. Specifically, a recent study [64] has reported how developers ignore flaky-test failures. Another study [54] found that when developers ignored flaky-test failures during a build, the deployed build experienced more crashes than builds that did not contain flaky-test failures. Harman and O'Hearn [31] have even suggested that all tests should be assumed flaky and that regression testing techniques should be improved to reduce flaky-test failures.

One prominent type of flaky tests is *order-dependent (OD) tests* [44, 47, 72]. An OD test is a test that passes or fails depending only on the order in which the test is run. Typically, the order in which OD tests pass is the order in which developers prefer to run the tests; we refer to this order as the *original order*. Running the tests in orders other than the original order may cause flaky-test failures from OD tests, which we refer to as *OD-test failures*.

Although OD tests may cause OD-test failures, OD tests can help developers run their tests faster by allowing the tests to share resources [6]. For example, one test may create a database that another test uses, allowing the latter test to run faster as an OD test instead of recreating the database. Every popular Java testing framework already permits developers to specify dependencies among tests, including JUnit [2–4], TestNG [13], DepUnit [10], Cucumber [9], and Spock [1]. In fact, as of May 2020, the JUnit annotations `@FixMethodOrder` and `@TestMethodOrder`, and the TestNG attributes `dependsOnMethods` and `dependsOnGroups` appear in over 197k Java files on GitHub.

Regression testing techniques should not assume that all tests are independent, since developers may still want OD tests. Specifically, these techniques should produce only test orders where each test has the same test outcome as it has when run in the original order. For example, if all of the tests pass in the original order, they should also pass in the order produced by a regression testing technique. However, traditional regression testing techniques may not achieve this goal if the test suite contains OD tests.

To understand how problematic the assumption of independent tests is to regression testing techniques as well as how necessary it is for these techniques to be dependent-test-aware, we conduct the first study on the impact of OD tests on traditional regression testing techniques. Based on prior literature, we implement 4 test prioritization, 6 test selection, and 2 test parallelization algorithms<sup>1</sup>. We apply each algorithm to 11 Java modules from 8 projects, obtained from a prior dataset of OD tests [44]. Each module contains tests written by developers, which we refer to as *human-written* tests, and at least one test in the module is an OD test. Due to the attractiveness of automatic test generation tools to reduce developers' testing efforts, we also use Randoop [53], a state-of-the-art test generation tool, to obtain *automatically-generated* tests for each module. For both types of tests, all 12 regression testing algorithms produce orders that cause OD-test failures from tests that pass in the original order. Our findings provide empirical evidence that these regression testing algorithms should not ignore test dependencies.

We propose a new, general approach to enhance traditional regression testing algorithms to make them dependent-test-aware. Similar to unenhanced algorithms, our enhanced algorithms take as input the necessary metadata for the algorithms (e.g., coverage information for a test prioritization algorithm) and the original order. Besides the metadata, our enhanced algorithms also take as input a set of test dependencies (e.g., test  $t_1$  should be run only after running test  $t_2$ ). Our general approach enhances traditional algorithms by first using the traditional algorithms to produce an order, and then reordering or adding tests to the order such that any OD test is ordered and selected while satisfying the test dependencies.

<sup>1</sup>We refer to an algorithm as a specific implementation of a regression testing technique.

We evaluate our general approach by applying it to 12 regression testing algorithms and comparing the orders produced by the enhanced algorithms to those produced by the unenhanced ones. Specifically, we run both the enhanced and unenhanced algorithms on multiple versions of our evaluation projects. To evaluate our enhanced algorithms, we use DTDetector [72] to automatically compute a set of test dependencies. Our use of an automated tool to compute test dependencies demonstrates that even if developers are not manually specifying test dependencies in their projects now, they can still likely achieve the results of our work by using automated tools such as DTDetector. Ideally, one would automatically compute test dependencies frequently, so that they are up-to-date and orders produced with them would not cause OD-test failures. However, such an ideal is often infeasible because automatically computing these test dependencies can take substantial time. To imitate how developers may infrequently recompute test dependencies, for each of our evaluation projects we compute a set of test dependencies on one version and use them for future versions.

Our evaluation finds that the orders from the enhanced algorithms cause substantially fewer OD-test failures and run only marginally slower than the orders from the unenhanced algorithms. Although our enhanced algorithms may not be using the most up-to-date test dependencies, our evaluation still finds that the algorithms produce orders that have 80% fewer OD-test failures than the orders produced by the unenhanced algorithms. Furthermore, although our enhanced algorithms may add tests for test selection and parallelization, we find that the orders with extra tests run only 1% slower on average. Our results suggest that making regression testing algorithms dependent-test-aware can substantially reduce flaky-test failures with only a minor slowdown to run the tests.

This paper makes the following main contributions:

**Study.** A study of how OD tests affect traditional regression testing techniques such as test prioritization, test selection, and test parallelization. When we apply regression testing techniques to test suites containing OD tests, 82% of the human-written test suites and 100% of the automatically-generated test suites have at least one OD-test failure.

**Approach.** A general approach to enhance traditional regression testing techniques to be dependent-test-aware. We apply our general approach to 12 traditional, regression testing algorithms, and make them and our approach publicly available [8].

**Evaluation.** An evaluation of 12 traditional, regression testing algorithms enhanced with our approach, showing that the orders produced by the enhanced algorithms can have 80% fewer OD-test failures, while being only 1% slower than the orders produced by the unenhanced algorithms.

## 2 IMPACT OF DEPENDENT TESTS

To understand how often traditional regression testing techniques lead to flaky-test failures due to OD tests, denoted as OD-test failures, we evaluate a total of 12 algorithms from three well-known regression testing techniques on 11 Java modules from 8 real-world projects with test suites that contain OD tests.

**Table 1: Four evaluated test prioritization algorithms from prior work [24].**

Label	Ordered by
T1	Total statement coverage of each test
T2	Additional statement coverage of each test
T3	Total method coverage of each test
T4	Additional method coverage of each test

**Table 2: Six evaluated test selection algorithms. The algorithms select a test if it covers new, deleted, or modified coverage elements. We also consider reordering the tests after selection. These algorithms form the basis of many other test selection algorithms [20, 51, 52, 55, 70].**

Label	Selection granularity	Ordered by
S1	Statement	Test ID (no reordering)
S2	Statement	Total statement coverage of each test
S3	Statement	Additional statement coverage of each test
S4	Method	Test ID (no reordering)
S5	Method	Total method coverage of each test
S6	Method	Additional method coverage of each test

**Table 3: Two evaluated test parallelization algorithms. These algorithms are supported in industrial-strength tools [7].**

Label	Algorithm description
P1	Parallelize on test ID
P2	Parallelize on test execution time

## 2.1 Traditional Regression Testing Techniques

Test prioritization, selection, and parallelization are traditional regression testing techniques that aim to detect faults faster than simply running all of the tests in the given test suite. We refer to the order in which developers typically run all of these tests as the *original order*. These traditional regression testing techniques produce orders (permutations of a subset of tests from the original order) that may not satisfy test dependencies.

**2.1.1 Test Prioritization.** Test prioritization aims to produce an order for running tests that would fail and indicate a fault sooner than later [67]. Prior work [24] proposed test prioritization algorithms that reorder tests based on their (1) total coverage of code components (e.g., statements, methods) and (2) additional coverage of code components *not* previously covered. These algorithms typically take as input coverage information from a prior version of the code and test suite, and they use that information to reorder the tests on future versions<sup>2</sup>. We evaluate 4 test prioritization algorithms proposed in prior work [24]. Table 1 gives a concise description of each algorithm. Namely, the algorithms reorder tests such that the ones with more total coverage of code components (statements or methods) are run earlier, or reorder tests with more additional coverage of code components not previously covered to run earlier.

**2.1.2 Test Selection.** Test selection aims to select and run a subsuite of a program’s tests after every change, but detect the same faults

<sup>2</sup>There is typically no point collecting coverage information on a future version to reorder and run tests on that version, because collecting coverage information requires one to run the tests already.

as if the full test suite is run [67]. We evaluate 6 test selection algorithms that select tests based on their coverage of modified code components [20, 32]; Table 2 gives a concise description of each algorithm. The algorithms use program analysis to select every test that may be affected by recent code modifications [32]. Each algorithm first builds a control-flow graph (CFG) for the then-current version of the program  $P_{\text{old}}$ , runs  $P_{\text{old}}$ ’s test suite, and maps each test to the set of CFG edges covered by the test. When the program is modified to  $P_{\text{new}}$ , the algorithm builds  $P_{\text{new}}$ ’s CFG and then selects the tests that cover “dangerous” edges: program points where  $P_{\text{old}}$  and  $P_{\text{new}}$ ’s CFGs differ. We choose to select based on two levels of code-component granularity traditionally evaluated before, namely statements and methods [67]. We then order the selected tests. Ordering by test ID (an integer representing the position of the test in the original order) essentially does no reordering, while the other orderings make the algorithm a combination of test selection followed by test prioritization.

**2.1.3 Test Parallelization.** Test parallelization schedules the input tests for execution across multiple machines to reduce test latency – the time to run all tests. Two popular automated approaches for test parallelization are to parallelize a test suite based on (1) test ID and (2) execution time from prior runs [7]. A test ID is an integer representing the position of the test in the original order. We evaluate one test parallelization algorithm based on each approach (as described in Table 3). The algorithm that parallelizes based on test ID schedules the  $i^{\text{th}}$  test on machine  $i \bmod k$ , where  $k$  is the number of available machines and  $i$  is the test ID. The algorithm that parallelizes based on the tests’ execution time (obtained from a prior execution) iteratively schedules each test on the machine that is expected to complete the earliest based on the tests already scheduled so far on that machine. We evaluate each test parallelization algorithm with  $k = 2, 4, 8$ , and 16 machines.

## 2.2 Evaluation Projects

Our evaluation projects consist of 11 modules from 8 Maven-based Java projects. These 11 modules are a subset of modules from the comprehensive version of a published dataset of flaky tests [44]. We include all of the modules (from the dataset) that contain OD tests, except for eight modules that we exclude because they are either incompatible with the tool that we use to compute coverage information or they contain OD tests where the developers have already specified test orderings in which the OD tests should run in. A list of which modules that we exclude from the dataset and the reasons for why we exclude them are on our website [8].

In addition to the existing human-written tests, we also evaluate automatically-generated tests. Automated test generation tools [21, 22, 25, 53, 73] are attractive because they reduce developers’ testing efforts. These tools typically generate tests by creating sequences of method calls into the code under test. Although the tests are meant to be generated independently from one another, these tools often do not enforce test independence because doing so can substantially increase the runtime of the tests (e.g., restarting the VM between each generated test). This optimization results in automatically-generated test suites occasionally containing OD tests. Given the increasing importance of automatically-generated tests in both research and industrial use, we also investigate them for OD tests.

**Table 4: Statistics of the projects used in our evaluation.**

ID	Project	LOC		# Tests		# OD tests		# Evaluation versions	Days between versions	
		Source	Tests	Human	Auto	Human	Auto		Average	Median
M1	apache/incubator-dubbo - m1	2394	2994	101	353	2 (2%)	0 (0%)	10	4	5
M2	- m2	167	1496	40	2857	3 (8%)	0 (0%)	10	25	18
M3	- m3	2716	1932	65	945	8 (12%)	0 (0%)	10	10	8
M4	- m4	198	1817	72	2210	14 (19%)	0 (0%)	6	32	43
M5	apache/struts	3015	1721	61	4190	4 (7%)	1 (<1%)	6	61	22
M6	dropwizard/dropwizard	1718	1489	70	639	1 (1%)	2 (<1%)	10	11	6
M7	elasticjob/elastic-job-lite	5323	7235	500	566	9 (2%)	4 (1%)	2	99	99
M8	jfree/jfreechart	93915	39944	2176	1233	1 (<1%)	0 (0%)	5	13	2
M9	kevinsawicki/http-request	1358	2647	160	4537	21 (13%)	0 (0%)	10	42	4
M10	undertow-io/undertow	4977	3325	49	967	6 (12%)	1 (<1%)	10	22	15
M11	wildfly/wildfly	7022	1931	78	140	42 (54%)	20 (14%)	10	37	19
Total / Average / Median		122803	66531	3372	18637	111 (3%)	28 (<1%)	89	25	8

Specifically, we use Randoop [53] version 3.1.5, a state-of-the-art random test generation tool. We configure Randoop to generate at most 5,000 tests for each evaluation project, and to drop tests that are subsumed by other tests (tests whose sequence of method calls is a subsequence of those in other tests).

### 2.3 Methodology

Regression testing algorithms analyze one version of code to obtain metadata such as coverage or time information for every test so that they can compute specific orders for future versions. For each of our evaluation projects, we treat the version of the project used in the published dataset [44] as the latest version in a sequence of versions. In our evaluation, we define a “version” for a project as a particular commit that has a change for the module containing the OD test, and the change consists of code changes to a Java file. Furthermore, the code must compile and all tests must pass through Maven. We go back at most 10 versions from this latest version to obtain the First Version (denoted as *firstVer*) of each project. We may not obtain 10 versions for an evaluation project if it does not have enough commits that satisfy our requirements, e.g., commits that are old may not be compilable anymore due to missing dependencies. We refer to each subsequent version after *firstVer* as a *subseqVer*. For our evaluation, we use *firstVer* to obtain the metadata for the regression testing algorithms, and we evaluate the use of such information on the *subseqVers*. For automatically-generated test suites, we generate the tests on *firstVer* and copy the tests to *subseqVers*. Any copied test that does not compile on a *subseqVer* is dropped from the test suite when run on that version.

Table 4 summarizes the information of each evaluation project. Column “LOC” is the number of non-comment, non-blank lines in the project’s source code and human-written tests as reported by sloc [11] for *firstVer* of each evaluation project. Column “# Tests” shows the number of human-written tests and those generated by Randoop [53] for *firstVer* of each evaluation project. Column “# Evaluation versions” shows the number of versions from *firstVer* to latest version that we use for our evaluation, and column “Days between versions” shows the average and median number of days between the versions that we use for our evaluation.

To evaluate how often OD tests fail when using test prioritization and test parallelization algorithms, we execute these algorithms on the version immediately following *firstVer*, called the Second Version (denoted as *secondVer*). For test selection, we execute the

algorithms on all versions after *firstVer* up to the latest version (the version from the dataset [44]). The versions that we use for each of our evaluation projects are available online [8]. For all of the algorithms, they may rank multiple tests the same (e.g., two tests cover the same statements or two tests take the same amount of time to run). To break ties, the algorithms deterministically sort tests based on their ordering in the original order. Therefore, with the same metadata for the tests, our the algorithms would always produce the same order.

For the evaluation of test prioritization algorithms, we count the number of OD tests that fail in the prioritized order on *secondVer*. For test selection, given the change between *firstVer* and the future versions, we count the number of unique OD tests that fail from the possibly reordered selected tests on all future versions. For test parallelization, we count the number of OD tests that fail in the parallelized order on any of the machines where tests are run on *secondVer*. All test orders are run three times and a test is counted as OD only if it consistently fails for all three runs. Note that we can just count failed tests as OD tests because we ensure that all tests pass in the original order of each version that we use.

Note that the general form of the OD test detection problem is NP-complete [72]. To get an approximation for the maximum number of OD-test failures with which the orders produced by regression testing algorithms can cause, we apply DTDetector [72] to randomize the test ordering for 100 times on *firstVer* and all *subseqVers*. We choose randomization because prior work [44, 72] found it to be the most effective strategy in terms of time cost when finding OD tests. The DTDetector tool is sound but incomplete, i.e., every OD test that DTDetector finds is a real OD test, but DTDetector is not guaranteed to find every OD test in the test suite. Thus, the reported number is a lower bound of the total number of OD tests. Column “# OD tests” in Table 4 reports the number of OD tests that DTDetector finds when it is run on all versions of our evaluation projects.

There are flaky tests that can pass or fail on the same version of code but are not OD tests (e.g., flaky tests due to concurrency). For all of the test suites, we run each test suite 100 times in its original order and record the tests that fail as flaky but not as OD tests. We use this set of non-order-dependent flaky tests to ensure that the tests that fail on versions after *firstVer* are likely OD tests and not other types of flaky tests.

## 2.4 Results

Table 5 and Table 6 summarize our results (parallelization is averaged across  $k = 2, 4, 8$ , and 16). The exact number of OD-test failures for each regression testing algorithm is available on our website [8]. In Table 5, each cell shows the percentage of unique OD tests that fail in all of the orders produced by the algorithms of a technique over the number of known OD tests for that specific evaluation project. Cells with a “n/a” represent test suites (of evaluation projects) that do not contain any OD tests according to DTDetector and the regression testing algorithms. The “Total” row shows the percentage of OD tests that fail across all evaluation projects per technique over all OD tests found by DTDetector. In Table 6, each cell shows the percentage of OD tests across all evaluation projects that fail per algorithm. The dependent tests that fail in any two cells of Table 6 may not be distinct from one another.

On average, 3% of human-written tests and <1% of automatically-generated tests are OD tests. Although the percentage of OD tests may be low, the effect that these tests have on regression testing algorithms is substantial. More specifically, almost every project’s human-written test suite has at least one OD-test failure in an order produced by one or more regression testing algorithms (the only exceptions are jfree/jfreechart (M8) and wildfly/wildfly (M11)). These OD-test failures waste developers’ time or delay the discovery of a real fault.

According to Table 6, it may seem that algorithms that order tests by Total coverage (T1, T3, S2, S5) always have fewer OD-test failures than their respective algorithms that order tests by Additional coverage (T2, T4, S3, S6), particularly for test prioritization algorithms. However, when we investigate the algorithm and module that best exhibit this difference, namely kevinsawicki/http-request’s (M9) test prioritization results, we find that this one case is largely responsible for the discrepancies that we see for the test prioritization algorithms. Specifically, M9 contains 0 OD-test failures for T1 and T3, but 14 and 24 OD-test failures for T2 and T4, respectively. All OD tests that fail for M9’s T2 and T4 would fail when one particular test is run before them; we refer to this test that runs before as the dependee test. The dependent tests all have similar coverage, so in the Additional orders, these tests are not consecutive and many of them come later in the orders. The one dependee test then comes inbetween some dependent tests, causing the ones that come later than the dependee test to fail. In the Total orders, the dependee test has lower coverage than the dependent tests and is always later in the orders. If we omit M9’s prioritization results, we no longer observe any substantial difference in the number of OD-test failures between the Total coverage and Additional coverage algorithms.

**2.4.1 Impact on Test Prioritization.** Test prioritization algorithms produce orders that cause OD-test failures for the human-written test suites in eight evaluation projects. For automatically-generated test suites, test prioritization algorithms produce orders that cause OD-test failures in three projects. Our findings suggest that orders produced by test prioritization algorithms are more likely to cause OD-test failures in automatically-generated test suites than human-written test suites. Overall, we find that test prioritization algorithms produce orders that cause 23% of the human-written OD tests and 54% of the automatically-generated OD tests to fail.

**Table 5: Percentage of OD tests that fail in orders produced by different regression testing techniques.**

ID	OD tests that fail (per evaluation project)							
	Prioritization		Selection		Parallelization			
	Human	Auto	Human	Auto	Human	Auto		
M1	50%	n/a	100%	n/a	50%	n/a		
M2	67%	n/a	67%	n/a	0%	n/a		
M3	12%	n/a	50%	n/a	0%	n/a		
M4	7%	n/a	14%	n/a	14%	n/a		
M5	0%	100%	25%	100%	75%	100%		
M6	100%	0%	0%	0%	0%	100%		
M7	44%	50%	0%	0%	0%	25%		
M8	0%	n/a	0%	n/a	0%	n/a		
M9	71%	n/a	71%	n/a	0%	n/a		
M10	17%	0%	17%	0%	0%	100%		
M11	0%	60%	0%	0%	0%	30%		
<b>Total</b>	23%	54%	24%	4%	5%	36%		

**Table 6: Percentage of OD tests that fail in orders produced by individual regression testing algorithms.**

Type	OD tests that fail (per algorithm)											
	Prioritization				Selection				Parallelization			
	T1	T2	T3	T4	S1	S2	S3	S4	S5	S6	P1	P2
Human	5%	25%	5%	20%	1%	28%	31%	1%	5%	9%	2%	5%
Auto	36%	43%	71%	25%	4%	4%	4%	4%	4%	4%	21%	64%

**2.4.2 Impact on Test Selection.** Test selection algorithms produce orders that cause OD-test failures for the human-written test suites in seven evaluation projects and for the automatically-generated test suites in one project. These test selection algorithms produce orders that cause 24% of the human-written OD tests and 4% of the automatically-generated OD tests to fail. The algorithms that do not reorder a test suite (S1 and S4) produce orders that cause fewer OD-test failures than the algorithms that do reorder. This finding suggests that while selecting tests itself is a factor, reordering tests is generally a more important factor that leads to OD-test failures.

**2.4.3 Impact on Test Parallelization.** Test parallelization algorithms produce orders that cause OD-test failures because the algorithms may schedule an OD test on a different machine than the test(s) that it depends on. We again find that the parallelization algorithm that reorders tests, P2, produces orders that cause more OD-test failures than P1. This result reaffirms our finding from Section 2.4.2 that reordering tests has a greater impact on OD-test failures than selecting tests. The percentages reported in Table 5 and Table 6 are calculated from the combined set of OD tests that fail due to parallelization algorithms for  $k = 2, 4, 8$ , and 16 machines. The orders of these algorithms cause 5% of the human-written OD tests and 36% of the automatically-generated OD tests to fail on average.

## 2.5 Findings

Our study suggests the following two main findings.

**(1) Regression testing algorithms that reorder tests in the given test suite are more likely to experience OD test failures than algorithms that do not reorder tests in the test suite.** We see this effect both by comparing test selection algorithms that do not reorder tests (S1 and S4) to those that do, as well as comparing a test parallelization algorithm, P2, which does reorder tests, to P1, which does not. Developers using algorithms that reorder tests

would especially benefit from our dependent-test-aware algorithms described in Section 3.

**(2) Human-written and automatically-generated test suites are likely to fail due to test dependencies.** As shown in Table 5, we find that regression testing algorithms produce orders that cause OD-test failures in 82% (9 / 11) of human-written test suites with OD tests, compared to the 100% (5 / 5) of automatically-generated test suites. Both percentages are substantial and showcase the likelihood of OD-test failures when using traditional regression testing algorithms that are unaware of OD tests.

### 3 DEPENDENT-TEST-AWARE REGRESSION TESTING TECHNIQUES

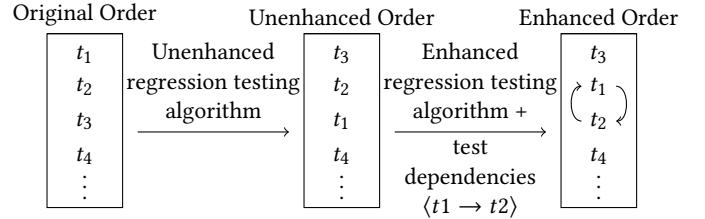
When a developer conducts regression testing, there are two options: running the tests in the original order or running the tests in the order produced by a regression testing algorithm. When the developer runs the tests in the original order, a test might fail because either (1) there is a fault somewhere in the program under test, or (2) the test is flaky but it is not a OD test (e.g., flaky due to concurrency). However, when using a traditional regression testing algorithm, there is a third reason for why tests might fail: the produced orders may not be satisfying the test dependencies. Algorithms that are susceptible to this third reason do not adhere to a primary design goal of regression testing algorithms. Specifically, the orders produced by these algorithms should not cause OD-test failures: if the tests all pass in the original order, then the algorithms should produce only orders in which all of the tests pass (and dually, if tests fail in the original order, then the algorithms should produce only orders in which these tests fail).<sup>3</sup> Since many real-world test suites contain OD tests, a regression testing algorithm that assumes its input contains no OD tests can produce orders that cause OD-test failures, violating this primary design goal (which we see from our results in Section 2).

We propose that regression testing techniques should be dependent-test-aware to remove OD-test failures. Our general approach completely removes all possible OD-test failures with respect to the input test dependencies. In practice, such test dependencies may or may not always be complete (all test dependencies that prevent OD-test failures are provided) or minimal (only test dependencies that are needed to prevent OD-test failures are provided). Nevertheless, our general approach requires as input an original order, a set of test dependencies, and an order outputted by a traditional regression testing algorithm to output an updated, enhanced order that satisfies the given test dependencies.

#### 3.1 Example

Figure 1 shows an illustrated example of the orders produced by an unenhanced algorithm and its corresponding enhanced algorithm. The unenhanced algorithm does not take test dependencies into account, and therefore may produce orders that cause OD-test failures. Specifically, the unenhanced algorithm produces the Unenhanced Order. On the other hand, the enhanced algorithm produces the Enhanced Order, a test order that satisfies the provided test dependencies of the test suite. The enhanced algorithm does so by

<sup>3</sup>Another design goal is to maximize fault-finding ability over time, i.e., efficiency. The efficiency goal is a trade-off against the correctness goal.



**Figure 1: Example of an unenhanced and its enhanced, dependent-test-aware regression testing algorithms.**

first using the unenhanced algorithm to produce the Unenhanced Order and then enforcing the test dependencies on to the order by reordering or adding tests.

We define two different types of test dependencies, positive and negative dependencies. A *positive test dependency*  $\langle p \rightarrow d \rangle$  denotes that for OD test  $d$  to pass, it should be run only after running test  $p$ , the test that  $d$  depends on. A *negative test dependency*  $\langle n \not\rightarrow d \rangle$  denotes that for OD test  $d$  to pass, it should *not* be run after test  $n$ . Previous work [61] refers to test  $p$  as a state-setter and test  $d$  in a positive dependency as a brittle. It also refers to test  $n$  as a polluter and  $d$  in a negative dependency as a victim. For simplicity, we refer to the tests that OD tests depend on (i.e.,  $p$  and  $n$ ) as *dependee tests*. For both types of dependencies, the dependee and OD test do not need to be run consecutively, but merely in an order that adheres to the specified dependencies.

In Figure 1, there is a single, positive test dependency  $\langle t_1 \rightarrow t_2 \rangle$  in the input test dependencies.  $\langle t_1 \rightarrow t_2 \rangle$  denotes that the OD test  $t_2$  should be run only after running test  $t_1$ . In the Unenhanced Order, the positive test dependency  $\langle t_1 \rightarrow t_2 \rangle$  is not satisfied and  $t_2$  will fail. Our enhanced algorithm prevents the OD-test failure of  $t_2$  by modifying the outputted order of the unenhanced algorithm so that the test dependency ( $t_2$  should be run only after running  $t_1$ ) is satisfied in the Enhanced Order.

#### 3.2 General Approach for Enhancing Regression Testing Algorithms

Figure 2 shows our general algorithm, `enhanceOrder`, for enhancing an order produced by a traditional regression testing algorithm to become dependent-test-aware. `enhanceOrder` takes as input  $T_u$ , which is the order produced by the traditional unenhanced algorithm that `enhanceOrder` is enhancing (this order can be a different permutation or subset of  $T_{orig}$ ), the set of test dependencies  $D$ , and the original test suite  $T_{orig}$ , which is an ordered list of tests in the original order. While in theory one could provide test dependencies that are not linearizable (e.g., both  $\langle t_1 \rightarrow t_2 \rangle$  and  $\langle t_2 \rightarrow t_1 \rangle$  in  $D$ ), we assume that the provided test dependencies are linearizable, and  $T_{orig}$  is the tests' one total order that satisfies all of the test dependencies in  $D$ . For this reason, `enhanceOrder` does not check for cycles within  $D$ . Based on  $T_u$ , `enhanceOrder` uses the described inputs to output a new order ( $T_e$ ) that satisfies the provided test dependencies  $D$ .

`enhanceOrder` starts with an empty enhanced order  $T_e$  and then adds each test in the unenhanced order  $T_u$  into  $T_e$  using the `addTest` function (Line 7). To do so, `enhanceOrder` first computes  $T_a$ , the set

```

enhanceOrder( $T_u, D, T_{orig}$ ):
1:  $T_e \leftarrow []$ 
2:  $P \leftarrow \{p \rightarrow d \in D\}$  // Positive test dependencies
3:  $N \leftarrow \{n \nrightarrow d \in D\}$  // Negative test dependencies
4:  $T_a \leftarrow T_u \circ (P^{-1})^*$  // Get all transitive positive dependee tests
5: for  $t : T_u$  do // Iterate  $T_u$  sequence in order
6:   if  $t \in T_e$  then continue end if
7:    $T_e \leftarrow \text{addTest}(t, T_e, T_u, T_{orig}, T_a, P, N)$ 
8: end for
9: return  $T_e$ 

addTest( $t, T_e, T_u, T_{orig}, T_a, P, N$ ):
10:  $B \leftarrow \{t' \in T_u \cup T_a | (t' \rightarrow t) \in P \vee (t \rightarrow t') \in N\}$ 
11:  $L \leftarrow \text{sort}(B \cap T_u, \text{orderBy}(T_u)) \oplus \text{sort}(B \setminus T_u, \text{orderBy}(T_{orig}))$ 
12: for  $b : L$  do // Iterate the before tests in order of  $L$ 
13:   if  $b \in T_e$  then continue end if
14:    $T_e \leftarrow \text{addTest}(b, T_e, T_u, T_{orig}, T_a, P, N)$ 
15: end for
16: return  $T_e \oplus [t]$ 

```

**Figure 2: General approach to enhance an order from traditional regression testing algorithms.**

of tests that the tests in  $T_u$  transitively depend on from the positive test dependencies (Line 4).  $T_a$  represents the tests that the traditional test selection or parallelization algorithms did not include in  $T_u$ , but they are needed for OD tests in  $T_u$  to pass. `enhanceOrder` then iterates through  $T_u$  in order (Line 5) to minimize the perturbations that it makes to the optimal order found by the traditional unenhanced algorithm. The `addTest` function adds a test  $t$  into the current  $T_e$  while ensuring that all of the provided test dependencies are satisfied. Once all of  $T_u$ 's tests are added into  $T_e$ , `enhanceOrder` returns  $T_e$ .

On a high-level, the function `addTest` has the precondition that all of the tests in the current enhanced order  $T_e$  have their test dependencies satisfied in  $T_e$ , and `addTest` has the postcondition that test  $t$  is added to the end of  $T_e$  (Line 16) and all tests in  $T_e$  still have their test dependencies satisfied. To satisfy these conditions, `addTest` starts by obtaining all of the tests that need to run before the input test  $t$  (Line 10), represented as the set of tests  $B$ .

The tests in  $B$  are all of the dependee tests within the positive test dependencies  $P$  for  $t$ , i.e., all tests  $p$  where  $\langle p \rightarrow t \rangle$  are in  $P$ . Note that these additional tests must come from either  $T_u$  or  $T_a$  (the additional tests that the traditional algorithm does not add to  $T_u$ ). Line 10 just includes into  $B$  the direct dependee tests of  $t$  and not those that it indirectly depends on; these indirect dependee tests are added to  $T_e$  through the recursive call to `addTest` (Line 14). The tests in  $B$  also include the dependent tests within the negative test dependencies  $N$  whose dependee test is  $t$ , i.e., all tests  $d$  where  $\langle t \rightarrow d \rangle$  are in  $N$ . `addTest` does not include test  $d$  that depends on  $t$  from the negative test dependencies if  $d$  is not in  $T_u$  or  $T_a$ . Conceptually, these are tests that the unenhanced algorithm originally did not find necessary to include (i.e., for test selection the test is not affected by the change, or for test parallelization the test is scheduled on another machine), and they are also not needed to prevent any OD tests already included in  $T_u$  from failing.

Once `addTest` obtains all of the tests that need to run before  $t$ , it then adds all of these tests into the enhanced order  $T_e$ , which `addTest` accomplishes by recursively calling `addTest` on each of these tests (Line 14). Line 11 first sorts these tests based on their order in the unenhanced order  $T_u$ . This sorting is to minimize the perturbations that it makes to the optimal order found by the unenhanced algorithm. For any additional tests not in  $T_u$  (tests added through  $T_a$ ), they are sorted to appear at the end and based on their order in the original order, providing a deterministic ordering for our evaluation (in principle, one can use any topological order). Once all of the tests that must run before  $t$  are included into  $T_e$ , `addTest` adds  $t$  (Line 16).

OD-test failures may still arise even when using an enhanced algorithm because the provided test dependencies ( $D$ ) may not be complete. For example, a developer may forget to manually specify some test dependencies, and even if the developer uses an automatic tool for computing test dependencies, such tool may not find all dependencies as prior work [72] has shown that computing *all* test dependencies is an NP-complete problem. Also, a developer may have made changes that invalidate some of the existing test dependencies or introduce new test dependencies, but the developer does not properly update the input test dependencies.

## 4 EVALUATION OF GENERAL APPROACH

Section 2 shows how both human-written and automatically-generated test suites with OD tests have OD-test failures when developers apply traditional, unenhanced regression testing algorithms on these test suites. To address this issue, we apply our general approach described in Section 3 to enhance 12 regression testing algorithms and evaluate them with the following metrics.

- **Effectiveness of reducing OD-test failures:** the reduction in the number of OD-test failures after using the enhanced algorithms. Ideally, every test should pass, because we confirm that all tests pass in the original order on the versions that we evaluate on (the same projects and versions from Section 2.2). This metric is the most important desideratum.
- **Efficiency of orders:** how much longer-running are orders produced by the enhanced regression testing algorithms than those produced by the unenhanced algorithms.

### 4.1 Methodology

To evaluate 12 enhanced regression testing algorithms, we start with *firstVer* for each of the evaluation projects described in Section 2.2. Compared to the unenhanced algorithms, the only additional input that the enhanced algorithms require is test dependencies  $D$ . These test dependencies  $D$  and the other metadata needed by the regression testing algorithms are computed on *firstVer* of each evaluation project. The details on how we compute test dependencies for our evaluation are in Section 4.2. With  $D$  and the other inputs required by the unenhanced and enhanced algorithms, we then evaluate these algorithms on *subseqVers* of the projects.

In between *firstVer* and *subseqVers* there may be tests that exist in one but not the other. We refer to tests that exist in *firstVer* as *old tests* and for tests that are introduced by developers in a future version as *new tests*. When running the *old tests* on future versions, we use the enhanced algorithms, which use coverage or

time information from *firstVer* (also needed by the unenhanced algorithms) and *D*. Specifically, for all of the algorithms, we use the following procedure to handle changes in tests between *firstVer* and a *subseqVer*.

- (1) The test in *firstVer* is skipped if *subseqVer* no longer contains the corresponding test.
- (2) Similar to most traditional regression testing algorithms, we treat tests with the same fully-qualified name in *firstVer* and *subseqVer* as the same test.
- (3) We ignore *new tests* (tests in *subseqVer* but not in *firstVer*), because both unenhanced and enhanced regression testing algorithms would treat these tests the same (i.e., run all of these tests before or after *old tests*).

## 4.2 Computing Test Dependencies

Developers can obtain test dependencies for a test suite by (1) manually specifying the test dependencies, or (2) using automatic tools to compute the test dependencies [26, 29, 44, 65, 72]. For our evaluation, we obtain test dependencies through the latter approach, using automatic tools, because we want to evaluate the scenario of how any developer can benefit from our enhanced algorithms without having to manually specify test dependencies. Among the automatic tools to compute test dependencies, both DTDetector [72] and iDFlakies [44] suggest that randomizing a test suite many times is the most cost effective way to compute test dependencies. For our evaluation, we choose to use DTDetector since it is the more widely cited work on computing test dependencies. Before we compute test dependencies, we first filter out tests that are flaky but are not OD tests (e.g., tests that are flaky due to concurrency [44, 47]) for each of our evaluation projects. We filter these tests by running each test suite 100 times in its original order and removing all tests that had test failures. We remove these tests since they can fail for other reasons and would have the same chance of affecting unenhanced and enhanced algorithms. To simulate how developers would compute test dependencies on a current version to use on future versions, we compute test dependencies on a prior version (*firstVer*) of a project’s test suite and use them with our enhanced algorithms on future versions (*subseqVers*).

**4.2.1 DTDetector.** DTDetector [72] is a tool that detects test dependencies by running a test suite in a variety of different orders and observing the changes in the test outcomes. DTDetector outputs a test dependency if it observes a test *t* to pass in one order (denoted as *po*) and fail in a different order (denoted as *fo*). Typically, *t* depends on either some tests that run before *t* in *po* to be dependee tests in a positive test dependency (some tests must always run before *t*), or some tests that run before *t* in *fo* to be dependee tests in a negative test dependency (some tests must always run after *t*). *t* can also have both a positive dependee test and a negative dependee test; this case is rather rare, and we do not observe such a case in our evaluation projects. DTDetector outputs the minimal set of test dependencies by delta-debugging [28, 68] the list of tests coming before *t* in *po* and *fo* to remove as many tests as possible, while still causing *t* to output the same test outcome.

When we use DTDetector directly as its authors intended, we find that DTDetector’s reordering strategies require many hours to

**Table 7: Average time in seconds to run the test suite and average time to compute test dependencies for an OD test. “Prioritization” and “Parallelization” show the average time per algorithm, while “All 6” shows the average time across all 6 algorithms. “-” denotes cases that have no OD test for all algorithms of a particular technique.**

ID	Suite run time		Time to precompute test dependencies					
	Human	Auto	Prioritization		Parallelization		All 6	
	Human	Auto	Human	Auto	Human	Auto	Human	Auto
M1	7.3	0.3	64	-	24	-	44	-
M2	0.4	0.2	95	-	-	-	95	-
M3	184.2	0.2	1769	-	-	-	1769	-
M4	0.1	0.2	19	-	13	-	17	-
M5	2.4	0.4	-	396	31	215	31	275
M6	4.1	0.3	75	-	-	29	75	29
M7	20.4	45.6	241	242	-	176	241	216
M8	1.2	1.3	-	-	-	-	-	-
M9	1.2	0.1	28	-	-	-	28	-
M10	19.9	0.8	157	-	-	39	157	39
M11	2.3	0.4	-	484	-	210	-	438

run and are designed to search for test dependencies by randomizing test orders; however, we are interested in test dependencies only in the orders that arise from regression testing algorithms. To address this problem, we extend DTDetector to compute test dependencies using the output of the regression testing algorithms. This strategy is in contrast to DTDetector’s default strategies, which compute test dependencies for a variety of orders that may not resemble the outputs of regression testing algorithms. Specifically, for test prioritization or test parallelization, we use the orders produced by their unenhanced algorithms. DTDetector will find test dependencies for any test that fails in these orders, since these tests now have a passing order (all tests must have passed in the original order) and a failing order. Using these two orders, DTDetector will minimize the list of tests before an OD test, and we would then use the minimized list as test dependencies for the enhanced algorithms. If the new test dependencies with the enhanced algorithms cause new failing OD tests, we repeat this process again until the orders for the enhanced algorithms no longer cause any OD-test failure.

For test selection, we simply combine and use all of the test dependencies that we find for the test prioritization and test parallelization algorithms. We use the other algorithms’ orders because it is difficult to predict test selection orders on future versions, i.e., the tests selected in one version will likely be different than the tests selected in another version. This methodology simulates what developers would do: they know what regression testing algorithm to use but do not know what changes they will make in the future.

**4.2.2 Time to Precompute Dependencies.** Developers should not compute test dependencies as they are performing regression testing. Instead, as we show in our evaluation, test dependencies can be collected on the current version and be reused later.

Developers can compute test dependencies infrequently and offline. Recomputing test dependencies can be beneficial if new test dependencies are needed or if existing test dependencies are no longer needed because of the developers’ recent changes. While the developers are working between versions  $v_i$  and  $v_{i+1}$ , they can use that time to compute test dependencies. Table 7 shows the time in seconds to compute the test dependencies that we use in our

**Table 8: Percentage of how many fewer OD-test failures occur in the test suites produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. Higher percentages indicate that the test suites by the enhanced algorithms have fewer OD-test failures.**

ID	% Reduction in OD-test failures						
	Prioritization		Selection		Parallelization		
	Human	Auto	Human	Auto	Human	Auto	
M1	100%	n/a	50%	n/a	100%	n/a	
M2	100%	n/a	100%	n/a	-	n/a	
M3	100%	n/a	-25%	n/a	-	n/a	
M4	100%	n/a	60%	n/a	100%	n/a	
M5	-	60%	0%	100%	100%	82%	
M6	100%	-	-	-	-	100%	
M7	100%	57%	-	-	-	12%	
M8	-	n/a	-	n/a	-	n/a	
M9	100%	n/a	92%	n/a	-	n/a	
M10	100%	-	100%	-	-	100%	
M11	-	56%	-	-	-	29%	
<b>Total</b>	<b>100%</b>	<b>57%</b>	<b>79%</b>	<b>100%</b>	<b>100%</b>	<b>66%</b>	

evaluation. The table shows the average time to compute test dependencies per OD test across all test prioritization or parallelization algorithms (for the columns under “Prioritization” and “Parallelization”, respectively), and the time under “All 6” is the average time to compute dependencies per OD test across all six test prioritization and parallelization algorithms (as explained in Section 4.2.1, the test dependencies that test selection uses are the combination of those from test prioritization and parallelization). The reported time includes the time for checks such as rerunning failing tests multiple times to ensure that it is actually an OD test (i.e., it is not flaky for other reasons) as to avoid the computation of non-existent dependencies.

Although the time to compute test dependencies is substantially more than the time to run the test suite, we can see from Tables 4 and 7 that the time between versions is still much more than the time to compute test dependencies. For example, while it takes about half an hour, on average, to compute test dependencies per OD test in M3’s human-written test suite, the average number of days between the versions of M3 is about 10 days, thus still giving developers substantial time in between versions to compute test dependencies. Although such a case does not occur in our evaluation, even if the time between  $v_i$  and  $v_{i+1}$  is less than the time to compute test dependencies, the computation can start running on  $v_i$  while traditional regression testing algorithms (that may waste developers’ time due to OD-test failures) can still run on  $v_{i+1}$ . Once computation finishes, the enhanced regression testing algorithms can start using the computed test dependencies starting at the current version of the code (e.g., version  $v_{i+n}$  when the computation starts on  $v_i$ ). As we show in Section 4.3, these test dependencies are still beneficial many versions after they are computed.

### 4.3 Reducing Failures

Table 8 shows the reduction in the number of OD-test failures from the orders produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. We denote cases that have no OD tests (as we find in Section 2.4) as “n/a”, and cases where the unenhanced algorithms do not produce an order that

causes any OD-test failures as “-”. Higher percentages indicate that the enhanced algorithms are more effective than the unenhanced algorithms at reducing OD-test failures.

Concerning human-written tests, we see that enhanced prioritization and parallelization algorithms are very effective at reducing the number of OD-test failures. In fact, the enhanced prioritization and parallelization algorithms reduce the number of OD-test failures by 100% across all of the evaluation projects. For test selection, the algorithms reduce the number of OD-test failures by 79%. This percentage is largely influenced by M3, where the enhanced selection orders surprisingly lead to *more* failures than the unenhanced orders as indicated by the negative number (-25%) in the table.

There are two main reasons for why an enhanced order can still have OD-test failures: (1) changes from later versions introduce new tests that are OD tests and are not in *firstVer*, and (2) the computed test dependencies from *firstVer* are incomplete; as such, the regression testing algorithms would have OD-test failures due to the missing test dependencies. In the case of (1), if this case were to happen, then both enhanced and unenhanced orders would be equally affected, and for our evaluation, we simply ignored all newly added tests. In the case of (2), it is possible that the test dependencies computed on *firstVer* are incomplete for the same OD tests on a new version, either because the missing test dependencies are not captured on *firstVer* or because the test initially is not an OD test in *firstVer* but becomes one due to newly introduced test dependencies in the new version. In fact, for M3’s human-written test selection results, we see that the reason for why the enhanced orders have more OD-test failures is that the enhanced orders in later versions expose a test dependency that is missing from what is computed on *firstVer*. As we describe in Section 4.2, to efficiently compute test dependencies on *firstVer*, we use only the orders of test prioritization and test parallelization instead of many random orders as done in some previous work [44, 72]. It is important to note that such a case occurs in our evaluation only for M3, but nonetheless, this case does demonstrate the challenge of computing test dependencies effectively and efficiently.

For automatically-generated tests, the enhanced algorithms are also quite effective at reducing OD-test failures, though they are not as effective as the enhanced algorithms for human-written tests. For test selection, only M5’s unenhanced orders have OD-test failures, and the enhanced orders completely remove all of the OD-test failures across all versions. Specifically, the OD test that fails is missing a positive dependee test, and the enhanced test selection algorithms would add in that missing positive dependee test into the selected tests, which prevents the OD-test failure. Similarly, the enhanced test parallelization algorithms also add in some missing positive dependee tests, leading to a reduction in OD-test failures. Once again though, there are still some OD-test failures because some test dependencies were not captured on *firstVer*.

In summary, we find that the orders produced by all of the enhanced regression testing algorithms collectively reduce the number of OD-test failures by 81% and 71% for human-written and automatically-generated tests, respectively. When considering both human-written and automatically-generated tests together, the enhanced regression testing algorithms produce orders that reduce the number of OD-test failures by 80% compared to the orders produced by the unenhanced algorithms.

**Table 9: Percentage slower that orders produced by the enhanced algorithms run compared to those produced by the unenhanced algorithms. Higher percentages indicate that orders produced by the enhanced algorithms are slower.**

ID	% Time Slowdown			
	Selection		Parallelization	
	Human	Auto	Human	Auto
M1	9%	=	16%	7%
M2	-1%	=	9%	-8%
M3	3%	=	0%	0%
M4	5%	=	-2%	-2%
M5	=	1%	-1%	-1%
M6	-1%	=	2%	-3%
M7	6%	=	1%	0%
M8	=	=	5%	4%
M9	11%	=	2%	3%
M10	-5%	=	2%	-14%
M11	=	=	-2%	-7%
<b>Total</b>	1%	1%	1%	0%

#### 4.4 Efficiency

To accommodate test dependencies, our enhanced regression testing algorithms may add extra tests to the orders produced by the unenhanced test selection or parallelization algorithms ( $T_a$  on Line 4 in Figure 2). The added tests can make the orders produced by the enhanced algorithms run slower than those produced by the unenhanced algorithms. Table 9 shows the slowdown of running the orders from the enhanced test selection and parallelization algorithms. We do not compare the time for orders where the enhanced and unenhanced algorithms produce the exact same orders (not only just running the same tests but also having the same ordering of the tests), since both orders should have the same running time modulo noise. We mark projects that have the same orders for enhanced and unenhanced with “=” in Table 9. For parallelization, we compare the runtime of the longest running subsuite from the enhanced algorithms to the runtime of the longest running subsuite from the unenhanced algorithms. For each of the projects in Table 9, we compute the percentage by summing up the runtimes for all of the enhanced orders in the project, subtracting the summed up runtimes for all of the unenhanced orders, and then dividing the summed up runtimes for all of the unenhanced orders. The overall percentage in the final row is computed the same way except we sum up the runtimes for the orders across all of the projects.

Overall, we see that the slowdown is rather small, and the small speedups (indicated by negative numbers) are mainly due to noise in the tests’ runtime. For test parallelization of automatically-generated tests, we do observe a few cases that do not appear to be due to noise in the tests’ runtime though. Specifically, we see that there are often speedups even when tests are added to satisfy test dependencies (e.g., M10). We find that in these cases the enhanced orders are faster because the OD-test failures encountered by the unenhanced orders actually slow down the test suite runtime more than the tests added to the enhanced orders that prevent the OD-test failures. This observation further demonstrates that avoiding OD-test failures is desirable, because doing so not only helps developers avoid having to debug non-existent faults in their changes, but can also potentially speed up test suite runtime.

The overall test suite runtime slowdown of the enhanced algorithms compared to the unenhanced ones is 1% across all of the orders produced and across all types of tests (human-written and automatically-generated) for all of the evaluation projects.

#### 4.5 Findings

Our evaluation suggests the following two main findings.

**Reducing Failures.** The enhanced regression testing algorithms can reduce the number of OD-test failures by 81% for human-written test suites. The enhanced algorithms are less effective for automatically-generated test suites, reducing OD-test failures by 71%, but the unenhanced algorithms for these test suites generally cause fewer OD-test failures. Across all regression testing algorithms, the enhanced algorithms produce orders that cause 80% fewer OD-test failures than the orders produced by the unenhanced algorithms.

**Efficiency.** Our enhanced algorithms produce orders that run only marginally slower than those produced by the unenhanced algorithms. Specifically, for test selection and test parallelization, the orders produced by the enhanced algorithms run only 1% slower than the orders produced by the unenhanced algorithms.

### 5 DISCUSSION

#### 5.1 General Approach vs. Customized Algorithms

In our work, we focus on a general approach for enhancing existing regression testing algorithms. Our approach works on any output of these existing regression testing algorithms to create an enhanced order that satisfies the provided test dependencies. While our approach works for many different algorithms that produce an order, it may not generate the most optimal order for the specific purpose of the regression testing algorithm being enhanced.

For example, we enhance the orders produced by test parallelization algorithms by adding in the missing tests for an OD test to pass on the machine where it is scheduled to run. A more customized test parallelization algorithm could consider the test dependencies as it decides which tests get scheduled to which machines. If the test dependencies are considered at this point during the test parallelization algorithms, then it could create faster, more optimized scheduling of tests across the machines. However, such an approach would be specific to test parallelization (and may even need to be specialized to each particular test parallelization algorithm) and may not generalize to other regression testing algorithms. Nevertheless, it can be worthwhile for future work to explore customized regression testing algorithms that accommodate test dependencies.

#### 5.2 Cost to Provide Test Dependencies

Developers may create OD tests purposefully to optimize test execution time by doing some expensive setup in one test and have that setup be shared with other, later-running tests. If developers are aware that they are creating such tests, then the human cost for providing test dependencies to our enhanced algorithms is low.

If developers are not purposefully creating OD tests, and they are unaware that they are creating OD tests, then it would be beneficial to rely on automated tools to discover such OD tests for them

and use the outputs of these tools for our enhanced algorithms, as we demonstrate in our evaluation. The cost to automatically compute test dependencies (machine cost) is cheaper than the cost for developers to investigate test failures (human cost). Herzig et al. [33] quantified human and machine cost. They reported that the cost to inspect one test failure for whether it is a flaky-test failure is \$9.60 on average, and the total cost of these inspections can be about \$7 million per year for products such as Microsoft Dynamics. They also reported that machines cost \$0.03 per minute. For our experiments, the longest time to compute test dependencies for an OD test is for M3, needing about half an hour, which equates to just about \$0.90.

### 5.3 Removing OD Tests

OD-test failures that do not indicate faults in changes are detrimental to developers in the long run, to the point that, if these failures are not handled properly, one might wonder why a developer does not just remove these OD tests entirely. However, it is important to note that these tests function exactly as they are intended (i.e., finding faults in the code under test) when they are run in the original order. Therefore, simply removing them would mean compromising the quality of the test suite to reduce OD-test failures, being often an unacceptable tradeoff. Removing OD tests is especially undesirable for developers who are purposefully writing them for the sake of faster testing [6], evident by the over 197k Java files (on GitHub) that use some JUnit annotations or TestNG attributes to control the ordering of tests as of May 2020. As such, we hope to provide support for accommodating OD tests not just for regression testing algorithms but for different testing tasks. We believe that our work on making regression testing algorithms dependent-test-aware is an important step in this direction.

### 5.4 Evaluation Metrics

In our evaluation, we focus on the reduction in the number of OD-test failures in enhanced orders over unenhanced orders as well as the potential increase in testing time due to the additional tests that we may need for test selection and test parallelization (Section 4.4). Concerning test prioritization algorithms, prior work commonly evaluates them using Average Percentage of Faults Detected (APFD) [56, 67]. Traditionally, researchers evaluate the quality of different test prioritization algorithms by seeding faults/mutants into the code under test, running the tests on the faulty code, and then mapping what tests detect which seeded fault/mutant. To compare the orders produced by the different test prioritization algorithms, researchers would measure APFD for each order, which represents how early an order has a test that detects each fault/mutant.

In our work, we use real-world software that does not have failing tests due to faults in the code, ensured by choosing versions where the tests pass in the original order (Section 2.3). We do not seed faults/mutants as we want to capture the effects of real software evolution. As such, we do not and cannot measure APFD because there would be no test failures due to faults in the code under test; APFD would simply be undefined in such cases. Any test failures that we observe would be either OD-test failures or test failures due to other sources of flakiness.

## 6 THREATS TO VALIDITY

A threat to validity is that our evaluation considers only 11 modules from 8 Java projects. These modules may not be representative causing our results not to generalize. Our approach might behave differently on different programs, such as ones from different application domains or those not written in Java.

Another threat to validity is our choice of 12 traditional regression testing algorithms. Future work could evaluate other algorithms based on static code analysis, system models, history of known faults, and test execution results, and so forth. Future work could also enhance other techniques that run tests out of order, such as mutation testing [59, 69, 70], test factoring [23, 58, 66], and experimental debugging techniques [63, 68, 71].

Another threat is the presence of non-order-dependent flaky tests when we compute test dependencies. Non-order-dependent tests may also affect the metrics that we use for the regression testing algorithms (e.g., coverage and timing of tests can be flaky), thereby affecting the produced orders. We mitigate this threat by filtering out non-order-dependent flaky tests through the rerunning of tests in the original order. We also suggest that developers use tools [44, 60] to identify these tests and remove or fix them; a developer does not gain much from a test whose failures they would ignore. In future work, we plan to evaluate the impact of these issues and to improve our algorithms to directly handle these non-order-dependent tests.

## 7 RELATED WORK

### 7.1 Test Dependence Definitions and Studies

Treating test suites as sets of tests [34] and assuming test independence is common practice in the testing literature [20, 32, 35, 37, 38, 40, 41, 50–52, 56, 57, 62, 63, 70, 71]. Little prior research considered test dependencies in designing or evaluating testing techniques.

Bergelson and Exman [18] described a form of test dependencies informally: given two tests that each pass, the composite execution of these tests may still fail. That is, if  $t_1$  and  $t_2$  executed by themselves pass, executing the sequence  $\langle t_1, t_2 \rangle$  in the same context may fail. In the context of databases, Kapfhammer and Soffa [39] formally defined independent test suites and distinguished them from other suites. However, this definition focuses on program and database states that may not affect actual test outcomes [15, 16, 39]. We use a definition of OD test [72] based on test outcomes. Huo and Clause [36] studied assertions that depend on inputs not controlled by the tests themselves.

Some prior work [14, 27, 47] studied the characteristics of flaky tests – tests that have non-deterministic outcomes. Test dependencies do not imply non-determinism: a test may non-deterministically pass/fail without being affected by any other test. Non-determinism does not imply test dependencies: a program may have no sources of non-determinism, but two of its tests can be dependent. One line of work that mentions test determinism as an assumption defines it too narrowly, with respect to threads but ignoring code interactions with the external environment [32, 52]. Furthermore, a test may deterministically pass/fail even if it performs non-deterministic operations. Unlike our work in this paper, the preceding prior work neither evaluated the impact of OD-test failures on regression testing techniques, nor proposed an approach to accommodate them.

## 7.2 Techniques to Detect Test Dependencies

Various techniques have been proposed to detect test dependencies automatically. PolDet [29] finds tests that pollute the shared state — tests that modify some location on the heap shared across tests or on the file system. However, the tests that PolDet detects do not fail themselves, and while they pollute shared state, no other tests currently in the test suite may fail due to that polluted shared state. In contrast, we use DTDetector [72], which not only dynamically identifies dependee polluting tests but also identifies the tests that depend on and fail due to the polluted state (OD tests). DTDetector can also identify test dependencies caused by other factors, such as database and network access. Biagiola et al. [19] studied test dependencies within web applications, noting the challenges in tracking the test dependencies in shared state between server- and client-side parts of web applications. They proposed a technique for detecting these test dependencies based on string analysis and natural language processing. Waterloo et al. [65] built a static analysis tool to detect inter-test and external dependencies. Electric Test [17] over-approximates test dependencies by analyzing data dependencies of tests. The tool is not publicly available, but the authors informed us that the work was continued by Gambi et al. into PRADET [26], which also outputs test dependencies. We have tried using PRADET but encountered issues with it on some of our projects. Lam et al. [44] released iDFlakies, a toolset similar to DTDetector. Both toolsets come with similar strategies, namely to run tests in different orders as to detect OD tests. We choose to use DTDetector instead of iDFlakies since DTDetector is the more widely cited work. Lam et al. also released a dataset of flaky tests, with ~50% OD tests and we use their dataset for our evaluation.

## 7.3 Techniques for Managing Flaky Tests

Only a few techniques and tools have been developed to prevent or accommodate the impact of test dependence. Some testing frameworks provide mechanisms for developers to specify the order in which tests must run. For tests written in Java, JUnit since version 4.11 supports executing tests in lexicographic order by test method name [5], while TestNG [12] supports execution policies that respect programmer-written dependence annotations. Other frameworks such as DepUnit [10], Cucumber [9], and Spock [1] also provide similar mechanisms for developers to manually define test dependencies. These test dependencies specified by developers could be used directly by our general approach, or to improve the test dependencies computed using automatic tools (by adding missing or removing unnecessary test dependencies). Haidry and Miller [30] proposed test prioritization that prioritizes tests based on their number of OD tests, hypothesizing that running tests with more test dependencies is more likely to expose faults. In our work, we enhance traditional test prioritization algorithms (along with other regression testing algorithms) to satisfy test dependencies.

VMVM [16] is a technique for accommodating test dependencies through a modified runtime environment. VMVM's runtime resets the reachable shared state (namely, the parts of the in-memory heap reachable from static variables in Java) between test runs. The restoration is all done within one JVM execution of all tests, providing benefits of isolation per test without needing to start/stop separate JVMs per test. Similar to the work by Kapfhammer and

Soffa [39], VMVM considers a test as an OD test if it accesses a memory location that has been written by another test, being neither necessary nor sufficient to affect the test outcome. Note that VMVM does not aim to detect OD tests, and resets shared state for *all* tests, regardless of pollution or not. In our work, we focus on coping with OD tests run in a single, standard JVM, and we propose enhanced regression testing algorithms to accommodate test dependencies as to reduce OD-test failures. Nemo [46] is a tool that considers test dependencies for test suite minimization. Our work aims at different regression testing techniques and can be used in conjunction with Nemo. iFixFlakies [61] is a tool for automatically fixing OD tests. iFixFlakies relies on finding code from “helpers”, which are tests in the existing test suite that prevent OD-test failures when run before an OD test, to fix the OD tests. However, iFixFlakies cannot fix OD tests if these OD tests have no helpers. Developers would still need to use our enhanced algorithms that avoid OD-test failures even when there are no helpers or when they plan on using test dependencies to run their tests faster. Lam et al. [43] presented FaTB, a technique for accommodating Async Wait-related flaky tests. FaTB automatically finds the time tests should wait for asynchronous method calls to reduce the test-runtime and frequency of flaky-test failures. Our enhanced algorithms accommodate OD tests, which are a different type of flaky tests than Async Wait-related ones.

## 8 CONCLUSION

Test suites often contain OD tests, but traditional regression testing techniques ignore test dependencies. In this work, we have empirically investigated the impact of OD tests on regression testing algorithms. Our evaluation results show that 12 traditional, dependent-test-unaware regression testing algorithms produce orders that cause OD-test failures in 82% of the human-written test suites and 100% of the automatically-generated test suites that contain OD tests. We have proposed a general approach that we then use to enhance the 12 regression testing algorithms so that they are dependent-test-aware. We have made these 12 algorithms and our general approach publicly available [8]. Developers can use the enhanced algorithms with test dependencies manually provided or automatically computed using various tools, and the enhanced algorithms are highly effective in reducing the number of OD-test failures. Our proposed enhanced algorithms produce orders that result in 80% fewer OD-test failures, while being 1% slower to run than the unenhanced algorithms.

## ACKNOWLEDGMENTS

We thank Jonathan Bell, Sandy Kaplan, Martin Kellogg, Darko Marinov, and the anonymous referees who provided feedback on our paper. This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0107, AFRL FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also partially supported by NSF grant numbers CNS-1564274, CNS-1646305, CNS-1740916, CCF-1763788, CCF-1816615, and OAC-1839010. Tao Xie is affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

## REFERENCES

- [1] 2011. Spock Stepwise. <https://www.canoo.com/blog/2011/04/12/spock-stepwise.html>.
- [2] 2012. JUnit and Java 7. <http://intellijava.blogspot.com/2012/05/junit-and-java-7.html>.
- [3] 2013. JUnit test method ordering. <http://www.java-allandsundry.com/2013/01/04/junit-test-method-ordering.html>.
- [4] 2013. Maintaining the order of JUnit3 tests with JDK 1.7. <https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK>.
- [5] 2013. Test execution order in JUnit. <https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.11.md#test-execution-order>.
- [6] 2016. Running your tests in a specific order. <https://www.ontestautomation.com/running-your-tests-in-a-specific-order>
- [7] 2019. Run tests in parallel using the Visual Studio Test task. <https://docs.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-vstest>.
- [8] 2020. Accommodating Test Dependence Project Web. <https://sites.google.com/view/test-dependence-impact>
- [9] 2020. Cucumber Reference - Scenario hooks. <https://cucumber.io/docs/cucumber/api/#hooks>.
- [10] 2020. DepUnit. <https://www.openhub.net/p/depunit>.
- [11] 2020. SLOCCount. <https://d Wheeler.com/slocount>
- [12] 2020. TestNG. <http://testing.org>.
- [13] 2020. TestNG Dependencies. <https://testing.org/doc/documentation-main.html#dependent-methods>.
- [14] Stephan Arlt, Tobias Morciniec, Andreas Podelski, and Silke Wagner. 2015. If A fails, can B still succeed? Inferring dependencies between test results in automotive system testing. In *ICST*. Graz, Austria, 1–10.
- [15] Jonathan Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *FSE*. Hong Kong, 799–802.
- [16] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*. Hyderabad, India, 550–561.
- [17] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*. Bergamo, Italy, 770–781.
- [18] Benny Bergelson and Iaakov Exman. 2006. Dynamic test composition in hierarchical software testing. In *Convention of Electrical and Electronics Engineers in Israel*. Eilat, Israel, 37–41.
- [19] Matteo Biagioli, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *ESEC/FSE*. Tallinn, Estonia, 154–164.
- [20] Lionel C. Briand, Yvan Labiche, and S. He. 2009. Automating regression test selection based on UML designs. *Information and Software Technology* 51, 1 (January 2009), 16–30.
- [21] Koen Claessen and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*. Montreal, Canada, 268–279.
- [22] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (September 2004), 1025–1050.
- [23] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *FSE*. Portland, OR, USA, 253–264.
- [24] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *ISSTA*. Portland, OR, USA, 102–112.
- [25] Gordon Fraser and Andreas Zeller. 2011. Generating parameterized unit tests. In *ISSTA*. Toronto, Canada, 364–374.
- [26] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*. Västerås, Sweden, 1–11.
- [27] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*. Florence, Italy, 55–65.
- [28] Alex Groce, Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *ICST*. Cleveland, OH, USA, 243–252.
- [29] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*. Baltimore, MD, USA, 223–233.
- [30] Shifa Zahra Haidry and Tim Miller. 2013. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering* 39, 2 (2013), 258–275.
- [31] Mark Harman and Peter O’Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*. 1–23.
- [32] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujrathi. 2001. Regression test selection for Java software. In *OOPSLA*. Tampa Bay, FL, USA, 312–326.
- [33] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *ICSE*. Florence, Italy, 483–493.
- [34] William E. Howden. 1975. Methodology for the generation of program test data. *IEEE Transactions on Computers* C-24, 5 (May 1975), 554–560.
- [35] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *ICSE*. Vancouver, BC, Canada, 419–429.
- [36] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*. Hong Kong, 621–631.
- [37] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *ASE*. Auckland, NZ, 233–244.
- [38] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. Orlando, Florida, 467–477.
- [39] Gregory M. Kapfhammer and Mary Lou Soffa. 2003. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE*. Helsinki, Finland, 98–107.
- [40] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*. Orlando, Florida, 119–129.
- [41] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2013. Optimizing unit test execution in large software programs using dependency analysis. In *APSys*. Singapore, 19:1–19:6.
- [42] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhia, and Suresh Thummala. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*. Beijing, China, 101–111.
- [43] Wing Lam, Kivanc Muşlu, Hitesh Sajnani, and Suresh Thummala. 2020. A Study on the Lifecycle of Flaky Tests. In *ICSE*. Seoul, South Korea, pages-to-appear.
- [44] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakes: A framework for detecting and partially classifying flaky tests. In *ICST*. Xi'an, China, 312–322.
- [45] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *ICSE*. Gothenburg, Sweden, 688–698.
- [46] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *ICSE*. Gothenburg, Sweden, 1039–1049.
- [47] Qingzhou Luo, Farah Hariri, Lamaya Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*. Hong Kong, 643–653.
- [48] John Micco. 2016. Flaky tests at Google and how we mitigate them. <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>
- [49] John Micco. 2017. The state of continuous integration testing @ Google. <https://ai.google/research/pubs/pub45880>
- [50] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel test generation and execution with Korat. In *ESEC/FSE*. Dubrovnik, Croatia, 135–144.
- [51] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *ICST*. Berlin, Germany, 21–30.
- [52] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*. Newport Beach, CA, USA, 241–251.
- [53] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE*. Minneapolis, MN, USA, 75–84.
- [54] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*. Lake Buena Vista, FL, USA, 857–862.
- [55] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu. 2004. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology* 13, 3 (July 2004), 277–331.
- [56] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (October 2001), 929–948.
- [57] Matthew J. Rummel, Gregory M. Kapfhammer, and Andrew Thall. 2005. Towards the prioritization of regression test suites with data flow information. In *SAC*. Santa Fe, NM, USA, 1499–1504.
- [58] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for Java. In *ASE*. Long Beach, CA, USA, 114–123.
- [59] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *ISSTA*. Chicago, IL, USA, 69–80.
- [60] August Shi, Alex Gyori, Owolabi Legusen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. Chicago, IL, USA, 80–90.
- [61] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakes: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*. Tallinn, Estonia, 545–555.
- [62] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *ISSTA*. Rome, Italy, 97–106.
- [63] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*. Lugano, Switzerland, 314–324.
- [64] Swapna Thorve, Chandani Sreshttha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *ICSME*, *NIER Track*. Madrid, Spain, 534–538.

- [65] Matias Waterloo, Suzette Person, and Sebastian Elbaum. 2015. Test analysis: Searching for faults in tests. In *ASE*. Lincoln, NE, USA, 149–154.
- [66] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezhang Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. 2010. Language-based replay via data flow cut. In *FSE*. Santa Fe, NM, USA, 197–206.
- [67] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (March 2012), 67–120.
- [68] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 3 (February 2002), 183–200.
- [69] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*. Lugano, Switzerland, 235–245.
- [70] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *ISSTA*. Minneapolis, MN, USA, 331–341.
- [71] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. Indianapolis, IN, USA, 765–784.
- [72] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. San Jose, CA, USA, 385–396.
- [73] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined static and dynamic automated test generation. In *ISSTA*. Toronto, Canada, 353–363.

# Differential Regression Testing for REST APIs

Patrice Godefroid  
Microsoft Research  
United States  
pg@microsoft.com

Daniel Lehmann\*  
University of Stuttgart  
Germany  
mail@lehmann.eu

Marina Polishchuk  
Microsoft Research  
United States  
marinapo@microsoft.com

## ABSTRACT

Cloud services are programmatically accessed through REST APIs. Since REST APIs are constantly evolving, an important problem is how to prevent breaking changes of APIs, while supporting several different versions. To find such breaking changes in an automated way, we introduce *differential regression testing* for REST APIs. Our approach is based on two observations. First, breaking changes in REST APIs involve two software components, namely the client and the service. As such, there are also two types of regressions: *regressions in the API specification*, i.e., in the contract between the client and the service, and *regressions in the service itself*, i.e., previously working requests are “broken” in later versions of the service. Finding both kinds of regressions involves testing along two dimensions: when the service changes and when the specification changes. Second, to detect such bugs automatically, we employ *differential testing*. That is, we compare the behavior of different versions on the same inputs against each other, and find regressions in the observed differences. For generating inputs (sequences of HTTP requests) to services, we use RESTler, a stateful fuzzer for REST APIs. Comparing the outputs (HTTP responses) of a cloud service involves several challenges, like abstracting over minor differences, handling out-of-order requests, and non-determinism. Differential regression testing across 17 different versions of the widely-used Azure networking APIs deployed between 2016 and 2019 detected 14 regressions in total, 5 of those in the official API specifications and 9 regressions in the services themselves.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Correctness;
- Networks → Cloud computing.

## KEYWORDS

REST APIs, differential regression testing, service regression, specification regression, client/service version matrix

### ACM Reference Format:

Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential Regression Testing for REST APIs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397374>

\*The work of this author was mostly done at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397374>

'20), July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397374>

## 1 INTRODUCTION

Cloud computing is exploding. Today, most cloud services, such as those provided by Amazon Web Services (AWS) [8] and Microsoft Azure [37], are programmatically accessed through REST APIs [21], both by third-party applications [7] and other services [40]. Since cloud services and their REST APIs are constantly evolving, breaking changes between different versions of a particular API are a major problem [19, 30]. For instance, a recent study of 112 “high-severity Azure production incidents caused by software-bugs” points out that in many cases “these bugs are triggered by software updates” [32]. Our paper discusses a fundamental software-engineering problem: *How to find such regressions effectively through automatic testing for cloud services with published APIs?*

Today, there are two main techniques to detect regressions when updating a REST API. First, testing *before* deploying a new API version is used to send hand-written or previously-recorded requests to the new service and check for specific responses. This is a laborious task, even though supported by a variety of commercial testing tools [2–5]. Checking the new specification is often even less automated and more incomplete, e.g., if only done by visual inspection. The second technique, especially for critical services, is to roll-out in a staged manner, e.g., starting with early-adopter deployments, then to select regions or datacenters, and then to all regions of a public cloud. Service traffic is constantly monitored and if clients stop working, errors will appear in traffic logs, and customers will complain. But catching regressions *after* rolling out a new service version is *painful and expensive* because incident management involves complex tasks (incident triage, prioritization, root-cause analysis, remediation, bug fixes, rolling-out patches, etc.) and disrupts engineers both on the service-provider side and on the customer side. Customer-visible regressions reduce customer satisfaction, and may also have direct financial impact when SLAs (Service Level Agreements) are broken.

This paper introduces *differential regression testing for REST APIs* as an automated technique to detect breaking changes across API versions. A first key observation is that breaking changes in REST APIs involve *two* software components, namely the client and the service. As such, we observe that there are also two types of regressions: *regressions in the API specification*, i.e., in the contract between the client and the service, and *regressions in the service itself*, i.e., previously valid requests stop working in later versions of the service. Finding both kinds of regressions involves testing along two dimensions: when the service changes and when new clients are derived from the changed specification.

A second key idea is to use *differential testing* to detect specification and service regressions automatically. We compare the

behavior of different versions of client-service pairs, and find regressions in the observed differences. Making this approach practical involves overcoming several technical challenges. Given  $N$  API versions, there are  $N^2$  client-service pairs, and  $N^4$  ways to compare them with each other. Fortunately, we show that not all these combinations need be considered in practice. We also discuss how to compare the outputs (HTTP responses) of a cloud service that contain non-determinism, and how to handle out-of-order or newly appearing requests when updating the specification.

To evaluate the effectiveness of differential regression testing, we present a detailed historical analysis of 17 versions of Microsoft Azure networking APIs deployed between 2016 and 2019. All their API specifications are publicly available on GitHub [36], and all 17 service versions are still accessible to anyone with an Azure subscription. Our approach and tools detected 5 regressions in the official specifications and 9 regressions in the services themselves. We also discuss how these regressions were fixed in subsequent versions.

The main contributions of this paper are:

- We introduce *differential regression testing for REST APIs*, including the key notions of *service* and *specification* regressions. We discuss the computational complexity of finding such regressions across  $N$  API versions.
- We discuss *how* to effectively detect service and specification regressions by comparing network logs capturing REST API traffic, using *stateful* and *stateless* diffing techniques.
- We present a detailed *API history analysis* for Microsoft Azure Networking, a widely-used core Azure service composed of more than 30 APIs. We found 14 regressions across 17 versions of those APIs deployed between 2016 and 2019.

## 2 DIFFERENTIAL REGRESSION TESTING FOR REST APIs

### 2.1 Regression Testing and Differential Testing

Before we describe our approach for differential regression testing for REST APIs, we define here some common terminology used throughout the remainder of the paper.

A *regression* in a program is a bug which causes a feature that worked correctly to stop working after a certain event, such as a software update. An *update* is the process of changing the program from an old version  $n$  to a new version  $n + 1$ . Regressions are also called *breaking changes*. The notion of breaking change only makes sense when the software versions before and after the update are in a compatibility relation with each other. *Major versions* in semantic versioning [42] are the canonical example of versions without a compatibility relation, i.e., which allow breaking changes by design (and thus should occur as little as possible). We only test versions that are in a compatibility relation with each other. Usually, those allow for *backwards-compatible changes*: New functionality may be added, but program inputs that “worked correctly” should still behave the same, if given to the new version.

*Regression testing* typically consists of running manually written and over-the-years accumulated test cases that check that the new program does not crash and that assertions specified in the program or in the test harness are not violated. Note that (classical) regression testing does *not* involve any comparison of program outputs with

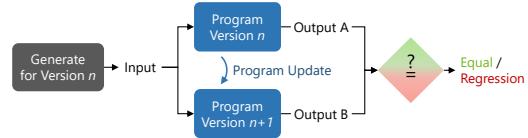


Figure 1: Differential regression testing for programs.

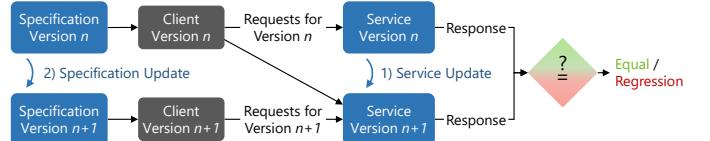


Figure 2: Differential regression testing for REST APIs involves two versions – for the specification and the service.

outputs from another run: each test of a regression test suite either passes or fails, and any failure indicates a potential regression (assuming all tests pass in the previous version).

*Differential testing* means comparing the outputs of two different, but related programs on the same input [34]. This notion is independent of the notion of regression testing. Differential testing is a generic method to obtain a test oracle [13] in automatic testing, i.e., to decide for generated inputs, whether the output of the program under test is correct. For instance, differential testing can be used to find bugs in compilers by compiling randomly-generated C programs using different compilers (e.g., clang and gcc), and then comparing the runtime behavior of the binaries produced by each compiler against each other [48].

*Differential regression testing* uses differential testing to automatically find regressions, and has been applied previously to “traditional” programs, e.g., compilers [25]. For it to work, one needs two backwards-compatible versions  $n$  and  $n + 1$  of the program under test and an input generator for said program. Differential regression testing then consists of the following steps, illustrated in Figure 1: both program versions  $n$  and  $n + 1$  are executed with the same, generated input valid for program  $n$ ; if the new version produces an output different from the previous version, a potential regression has been detected. We use differential regression testing for REST APIs. Below, we describe a crucial difference between traditional programs and REST APIs that presents additional challenges for differential regression testing.

### 2.2 Updates in REST APIs

A REST API is *not* a traditional program because it defines an *interface* between two software components: a *client*, which produces requests, and a *service*, which handles these requests and returns responses. Thus, while in traditional regression testing only a single program is updated, in the context of REST APIs, both the service and the clients are updated over time.

The contract between clients and the service is the API *specification*. It defines which requests clients may send, e.g., which HTTP methods (GET, PUT, etc.) are available for each endpoint (URI) and what is allowed in requests and responses. A common format for such specifications is Swagger (also known as OpenAPI) [46].

Developers write client code mostly based on this specification since the service itself is usually a black box (e.g., for commercial services, source code is not available). Given that the specification is authoritative information for client developers, it is crucial that it is correct. In many cases, whole clients are also automatically generated from specifications. For instance, the Swagger specifications of Microsoft Azure services are publicly available on GitHub [36], and software development kits (SDKs) for various languages like Python, JavaScript, or C# are auto-generated from them [6].

Because a REST API has two components – a client that is derived from a specification, and a service – Figure 2 illustrates that, unlike for regression testing of “traditional” programs, there is not just a single update happening, but in fact two orthogonal update steps:

- First, the *service* is updated to be able to handle added requests and functionality of the new API version. If the new API version is backwards-compatible with the old one, the service is also expected to handle existing requests correctly.
- Second, the updated *specification* is published so that developers know about the added functionality and can write new clients for the new service.

By publishing the updated specification, the service owner “commits” to the new API version – third parties can now write and generate their own clients to use the newly documented functionality. In other words, updating a REST API from a version  $n$  to a new version  $n + 1$  generates *two* new component versions: a new service version  $n + 1$ , denoted  $s_{n+1}$ , and a new client version  $n + 1$ , denoted  $c_{n+1}$ , corresponding to the new API specification.

In order to test a service, it is necessary to exercise it with a client. In what follows, the pair  $(c_i, s_j)$  will denote testing the service  $s_j$  with a client  $c_i$  matching (or automatically generated from) specification version  $i$ . Figure 2 shows the three possible testing configurations between the client/service versions  $n$  and  $n + 1$  of a REST API, which are each denoted by an edge in Figure 2:

- $(c_n, s_n)$  corresponds to the vertical edge where inputs from old clients are sent to an old service version,
- $(c_n, s_{n+1})$  (the diagonal edge) is the state when the service is already updated, but the new specification not yet published, and
- $(c_{n+1}, s_{n+1})$  is the final configuration when both clients and service are updated.

The hypothetical configuration  $(c_{n+1}, s_n)$  is not considered here because a new client version  $c_{n+1}$  is usually not supposed to work with the old service version  $s_n$  (unless the service is fully *forward-compatible*, but this is unlikely in practice and therefore this case will not be considered further in this paper). From now on, we call a pair  $(c_i, s_j)$  a *testing configuration*.

### 2.3 Regressions in REST APIs

Given *two* new component versions, one for the specification and clients derived from it, and another version for the service, there are now *two possible types of breaking changes*:

- (1) *Regressions in the service* occur when requests that worked with a previous version of the service are no longer accepted, or return different or wrong results.

- (2) *Regressions in the specification* occur when previously-allowed requests or responses are removed (or modified) in the new specification in a way that breaks applications after switching to the new client version.

Therefore, in order to detect regressions of both types, *two types of differential regression testing* are required for REST APIs:

- (1) Comparing the testing results of  $(c_n, s_n)$  with the results of  $(c_n, s_{n+1})$  may detect *regressions in the service* (the client version  $c_n$  stays constant).
- (2) Comparing the testing results of  $(c_n, s_{n+1})$  with the results of  $(c_{n+1}, s_{n+1})$  may detect *regressions in the specification* (the service version  $s_{n+1}$  stays constant).

This is what we call *differential regression testing for REST APIs*: how to compare the outputs of two different testing configurations in order to detect service or specification regressions in a REST API.

A typical example of *service regression* is when a request type suddenly stops working (which is rare), or when the format of the response unexpectedly changes and is undocumented (or incorrectly documented), for instance removing, adding or renaming response properties (which is more frequent). Examples of *specification regressions* are when an optional property in the body of a request becomes required, or when an item in an enumerated list is removed from the specification.

Since a specification and its service are closely related, one might think that regressions cannot appear independently in one or the other. However, in the context of cloud services, specifications and services are often authored and maintained by different people and written in different languages (e.g., the service can be written in C# and the specification in Swagger). So in practice, they can become inconsistent due to regressions in either of them.

Note that specification regressions may be *guessed by statically* comparing (diffing) specification descriptions. However, in order to *confirm* that such specification changes are either false alarms or actual regressions, it is *necessary to test* the new service with the old client in order to check whether the new service still supports the old functionality (now undocumented but perhaps still present for backwards compatibility), or is a true specification regression as defined above.

Do cloud services (like core Azure services) guarantee backwards compatibility, i.e., that the two types of regressions defined above should never occur? Strictly speaking no, but implicitly yes. Strictly speaking, clients are supposed to use the specification version  $n + 1$  with the service version  $n + 1$ . But in practice, backwards compatibility is implicitly expected as API versions change frequently (e.g., for Azure ca. every month), and continuous support of previous functionality is expected to allow past clients (customers) to continue operating and evolving their own services [24].

### 2.4 Client/Service REST API Version Matrix

The updates to the client and service versions defined by a REST API can be visualized using a two-dimensional *client/service version matrix*, as shown in Figure 3. The columns of this matrix correspond to service versions, while the rows correspond to client versions, or more accurately, to the versions of the API specifications from which clients can be generated. The cell in the matrix at row  $i$  and column  $j$  corresponds to testing  $(c_i, s_j)$ .

Client/Service Versions	$s_1$	$s_2$	$s_3$	$s_4$
$c_1$	$(c_1, s_1)$	$(c_1, s_2)$	$(c_1, s_3)$	$(c_1, s_4)$
$c_2$	$(c_2, s_1)$	$(c_2, s_2)$	$(c_2, s_3)$	$(c_2, s_4)$
$c_3$	$(c_3, s_1)$	$(c_3, s_2)$	$(c_3, s_3)$	$(c_3, s_4)$
$c_4$	$(c_4, s_1)$	$(c_4, s_2)$	$(c_4, s_3)$	$(c_4, s_4)$

**Figure 3: Example of a REST API version matrix, client versions increasing row-wise and service versions increasing column-wise. The dotted blue arrow marks an example of a service update. The dashed green arrow marks an example of a client update (derived from a new specification). Client/service configurations that are not tested are grayed-out.**

If  $N$  denotes the number of versions of an API, the client/service version matrix is of size  $N^2$ . However, some of these testing configurations do not make sense in our context: testing  $(c_i, s_j)$  when  $i > j$  is not necessary as *newer client* versions are not supposed to work with *older service* versions. In terms of the version matrix, this means the matrix is an *upper triangular matrix*. This reduces the number of testing configurations from  $N^2$  to  $N * (N + 1)/2$  (i.e., the number of cells in a triangular matrix, including its diagonal).

## 2.5 Complexity of Differential Regression Testing

Each cell of a client/service version matrix corresponds to one client-service testing configuration. An automated testing approach with a non-differential test oracle runs one test for each cell in the matrix. E.g., *RESTler* [12] can test that no request sent by a client  $c_i$  ever leads to a 500 Internal Server Error response by the service  $s_j$ . But to get a more interesting test oracle, we are employing *differential testing*, and thus have to ask how many of the test configurations should be *compared* with each other in order to find all possible regressions in  $N$  versions of an API? In other words, how many *pairs of cells* should be compared to achieve this goal?

When updating an API version from version  $n$  to  $n + 1$  for any  $1 \leq n < N$ , both service or specification regressions may be introduced, as defined in the previous subsection. Checking for service regressions by comparing the testing results of  $(c_n, s_n)$  with the results of  $(c_n, s_{n+1})$  corresponds to a horizontal edge in the version matrix as indicated by the blue dotted arrow in Figure 3. Similarly, checking for specification regressions by comparing the testing results of  $(c_n, s_n)$  with the results of  $(c_{n+1}, s_{n+1})$  corresponds to a vertical edge in the version matrix, highlighted as the green dashed arrow in Figure 3.

Instead of checking separately for service regressions and specification regressions, why not simply compare the testing results of  $(c_n, s_n)$  with  $(c_{n+1}, s_{n+1})$  directly? (Such comparisons would correspond to diagonal edges between cells in the version matrix.) The reason is that such comparisons would be harder, less accurate, and could miss detecting regressions. For example, if a request  $X$  is renamed to  $Y$  in version  $n + 1$ , both  $(c_n, s_n)$  using  $X$  and  $(c_{n+1}, s_{n+1})$  using  $Y$  may work fine (no errors), but automatically detecting that  $X$  has been renamed  $Y$  with high confidence (no false alarms) is

a harder problem. In contrast, leaving the client version  $c_n$  or the service version  $s_{n+1}$  constant avoids this “generalized mapping” problem, and facilitates the detection of service or specification regressions, respectively.

By generalization of the previous argument, it is also unnecessary to perform differential testing on any non-adjacent pairs of testing configurations in the version matrix: for any  $i' > i$  and  $j' > j$ , it is not necessary to compare the testing results of  $(c_i, s_j)$  with  $(c_{i'}, s_{j'})$ . Indeed, any service regression between the old  $(c_i, s_j)$  and the new  $(c_{i'}, s_{j'})$  will be found when differential testing of some  $(c_i, s_n)$  and  $(c_i, s_{n+1})$  with  $j \leq n < j'$ . Similarly, any specification regression between the old  $(c_i, s_j)$  and the new  $(c_i, s_{j'})$  will be found when differential testing of some  $(c_n, s_{j'})$  and  $(c_{n+1}, s_{j'})$  with  $i \leq n < i'$ . In other words, comparisons corresponding to *transitive edges* need not be considered in the version matrix: comparing only *atomic updates* is sufficient.

Consequently, for any testing configuration  $(c_i, s_j)$  in the version matrix, we need to compare it to *at most* its two adjacent neighbors along the two dimensions, namely  $(c_i, s_{j+1})$  (right neighbor) and  $(c_{i+1}, s_j)$  (bottom neighbor) if  $i < j$ .

If we want to cover all possible  $N * (N + 1)/2$  client-service  $(c_i, s_j)$  testing configurations (i.e., all the cells in the upper triangular version matrix including the main diagonal), the number of differential testing comparisons we need to perform is therefore:

- For every testing configuration on the main diagonal of the version matrix, one can only compare its output to the right neighbor (since the bottom neighbor is an invalid configuration of old service and new client):  $N$  comparisons.
- For every testing configuration in the rightmost column, one can only compare to bottom neighbors (since there are no more recent service versions to the right):  $N$  comparisons.
- For the bottom right corner (most recent client and service version), we performed two unnecessary comparisons:  $-2$ .
- For all other “interior cells”: 2 comparisons (one to the right, where the service is updated, one down, where the client is updated):  $\frac{(N-1)(N-2)}{2}$  cells times 2 comparisons per cell.
- This gives a total number of comparisons for differential testing:  $N + N - 2 + \frac{(N-1)(N-2)}{2} * 2 = N * (N - 1)$

In summary, given an API with  $N$  versions, there are  $T(N) = \frac{N*(N+1)}{2}$  possible client/service testing configurations, and finding regressions in all possible service and specification updates requires  $D(N) = N * (N - 1)$  comparisons (or diff's) between the outputs of those  $T(N)$  client/service configurations.

## 2.6 Diagonal-Only Strategy

Additionally, we also propose a simpler and less expensive strategy for finding service and specification regressions:

- For every new version  $n + 1$ , only two new testing configurations have to be run:  $(c_n, s_{n+1})$  and  $(c_{n+1}, s_{n+1})$ .
- Between the original configuration  $(c_n, s_n)$  and the two new configurations, perform only two comparisons: compare  $(c_n, s_n)$  with  $(c_n, s_{n+1})$  (to check for service regressions) and compare  $(c_n, s_{n+1})$  with  $(c_{n+1}, s_{n+1})$  (to check for specification regressions).

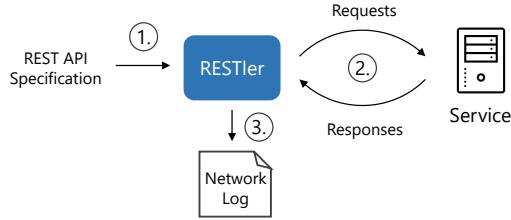


Figure 4: RESTler overview.

In other words, this strategy corresponds to only testing the configurations corresponding to the diagonal cells ( $c_{n+1}, s_{n+1}$ ) of the version matrix, plus all the cells ( $c_n, s_{n+1}$ ) immediately above these. Let us therefore call this strategy the *diagonal-only strategy*.

Given  $N$  API versions, the advantage of this strategy is that it requires only  $T_{\text{diag}}(N) = 2 * (N - 1)$  client-service test combinations, and only  $D_{\text{diag}}(N) = 2 * (N - 1)$  comparisons (diffs) among these. In theory, *assuming* services and specifications are backward-compatible, this strategy is as accurate as the upper-triangular matrix strategy, by transitivity of backward-compatibility, *if also assuming* a fixed deterministic test-generation algorithm from specifications, and *assuming* services responses are deterministic. However, if any of these three assumptions are not satisfied, the diagonal-only strategy provides less coverage among all the possible client-service version pairs, and therefore could miss more regressions compared to the upper-triangular matrix strategy, simply because it runs fewer tests.

In practice, is the diagonal-only strategy sufficient to find many, or even all the service and specification regressions that are found by the upper-triangular matrix strategy? In Section 4, we will empirically answer this question for 17 versions of a large complex API of a core Microsoft Azure service. But first, we need to define how to exercise a client version  $c_i$  with a service version  $s_j$ , and then how to compare testing results obtained with various such combinations. These are the topics discussed in the next section.

### 3 TECHNICAL CHALLENGES

#### 3.1 From Specifications to Tests

In order to automatically generate client code from a REST API specification and comprehensively exercise a service under test, we use a recent automatic test generation tool *RESTler* [12]. As shown in Figure 4, *RESTler* generates and executes sequences of the requests defined in the API specification, while recording all requests and responses in a *network log*. *RESTler* supports the Swagger (recently renamed OpenAPI) [46] interface-description language for REST APIs. A Swagger specification describes how to access a cloud service through its REST API, including what requests the service can handle, what responses may be received, and the response format. Most Azure services have public Swagger API specifications available on GitHub [36]. A test suite automatically generated by *RESTler* attempts to cover as much as possible of the input Swagger specification, but full coverage is not guaranteed. This limitation is acceptable because the purpose of this work is *only* to detect specification or service regressions through automated testing, *not to prove* the absence of regressions (verification).

```

1 [ [ { // a pair of sent request and received response
2   "request": {
3     "method": "PUT",
4     "uri": ["blog", "post-42"],
5     "queryString": { "api-version": "2017" },
6     "headers": {
7       "Authorization": "token-afe2d391031afe...",
8       "Content-Type": "application/json",
9       ... }, // more headers
10    "body": { // body as JSON, not plain string
11      "title": "My first blogpost", // property-1
12      "content": "..." } //property-2
13  },
14  "response": {
15    "status": "201 Created",
16    "headers": {
17      "Content-Length": "566",
18      "Date": "Wed, 31 Jul 2019 18:00:00 GMT",
19      "Server": "Apache/2.4.1 (Unix)",
20      "x-request-id": "fae12396-4557-4755-bd1b-7380d40d033c",
21      ... },
22    "body": {
23      "id": "/blog/2019-07-31/post-42", //property-3
24      ...
25  },
26  ... // more requests and responses for this sequence
27 },
28 ... // more sequences of requests and responses
29 ]
  
```

Figure 5: Example of a network log.

```

log ::= requestSequence*
requestSequence ::= requestResponsePair*
requestResponsePair ::= (request, response?)

request ::= method, uri, headers, body
response ::= statusCode, statusDescription, headers, body

method ::= GET | PUT | DELETE | PATCH | ...
uri ::= path, queryString
path ::= pathComponent/*
queryString ::= (key, value)*

statusCode ::= 200 | 201 | ... | 400 | ... | 500
statusDescription ::= OK | Created | ... | Bad Request | ...
| Internal Server Error

headers ::= (key, value)*
key, value, pathComponent ∈ strings
body ∈ JSON values
  
```

Figure 6: Abstract syntax of network logs. Common symbols: \* means zero or more repetitions, ? is zero or one.

Since we use *RESTler* as a building block in our approach, this limitation is orthogonal to our approach. Future improvements to *RESTler* that exercise services more thoroughly could potentially lead to more regressions found by our approach as well.

#### 3.2 Network Logs

The network logs considered in this work are standard, and are not *RESTler* specific. REST API traffic consists of sequences of request-response pairs. An example of a request and its response are shown in Figure 5 in the JSON format. Each request consists of a *method*, a *uri*, *headers*, and a *body*. Each response consists of the HTTP *status code and description*, *headers*, and a *body*. Figure 6 shows the abstract syntax of network logs capturing REST API traffic.

Resources managed through a REST API typically need to be created first (with a PUT or POST request), then can be accessed (with a GET request) or updated (with a PATCH or POST request), and then deleted (with a DELETE request). Some resources require a parent resource in order to be created. For example, the request `PUT /blog/post-42?api-version=2017` shown in Figure 5 creates a new *blog post* with id `post-42`; the response includes `201 Created`, which means the new resource has been successfully created; only then a request, for example, `PUT /blog/post-42/comment/3?api-version=2017`, may be executed to add a comment (child resource) associated with that blog post (parent resource). *RESTler* is *stateful*: it can automatically generate such sequences of causally-related requests in order to exercise “deeper” parts of an API specification (like the `comment` part of the API in the above example).

We define a network log as a sequence of sequences of request-response pairs. Each subsequence represents a “testing session” of usually-related requests with their responses. Note that the structure of HTTP requests and responses is explicitly represented in JSON, e.g., headers are a map from names to values, and URL paths are a sequence of path components. Also note that if the body is of type `application/json` (as is common for REST APIs), it is saved in the log a structured manner and not just as a plain string.

### 3.3 Differing of Network Logs

A naïve implementation for comparing two network logs might use an off-the-shelf textual differencing tool or structural differencing tool such as `json-diff` [23]. There are several reasons why this approach does not work well for identifying potential regressions in network logs:

First, not every part of the exchanged HTTP messages is relevant for testing the behavior of a cloud service. For example, HTTP header fields such as `Authorization` are expected to contain different values in each log, since authentication tokens are refreshed at the start of each test run. Another example is the `Content-Length` field of the response, which can be ignored because it adds only noise to the comparison. (The length of a body will be different only if there already was a difference in the contents.)

Second, requests automatically generated by *RESTler* may contain randomly chosen concrete values, which do not indicate service behavior changes. For example, unique names are generated when creating resources, and those names can also appear in responses.

Third, some values in returned responses are non-deterministic values generated by the service, which also do not indicate a difference in behavior. Examples of such values that are out of the control of the client are the `Date` response header, unique IDs for requests and responses (used for debugging in case of a service error) such as the `x-request-id` or `Etag` headers, or service dependent dynamic values. (E.g., for a DNS service API, the exact nameservers returned for a domain can vary for load balancing purposes.)

Thus, prior to performing regression differencing, we perform an *abstraction* of the network logs that excludes such values from the subsequent diff. Abstractions work by matching certain values in the network logs against a regular expression and replacing them by (fresh) constants, in order to equate all of their (possibly different, but irrelevant) values. The abstractions we apply are:

- Replace the values of the `Date`, `Etag`, and `x-*request-id` headers by constant strings.

- Replace IDs that were randomly generated by *RESTler* with constant strings, e.g., `post-42` and `post-23` both match the regular expression `post-\d+` and would be replaced by `post-id`.
- A set of service-specific replacements, e.g., non-deterministic name servers, UUIDs, etc.

Abstractions are specified by a single command-line parameter, so effort is minimal and done once per API.

*Recursive Comparisons.* After having applied abstractions, we compare two network logs by recursively comparing their subelements. To start with, we compare the first sequence of requests in the first network log with the first sequence of requests in the second network log. For sequences of strings, we use an algorithm (such as Myers diff [38]) that minimizes the number of edits between two such sequences. In the general case (i.e., differencing two sequences of requests), we do not minimize the size of our diff. Instead, if the first sequence is shorter than the second sequence, we regard elements after the common prefix as insertions, whereas for the case where the first sequence is longer than the second sequence, we regard those elements as deleted. We leave further minimizing the diffs, e.g., by employing tree differencing algorithms like in the *GumTree* tool [20] to future work.

We then recursively apply this comparison in a bottom-up fashion on the structured network logs to obtain a full network log diff, which has the same structure as a single network log, only with intermediate *insert*, *delete*, and *edit* nodes, at all subtrees where the two network logs were not equal.

### 3.4 Stateless Differing of Network Logs

The differencing approach of the previous section enables a precise analysis of all differences in network logs from two testing configurations. However, even after applying the above abstractions, the following problems remain that cause irrelevant differences to be shown:

- *Sensitivity to ordering and deletions/insertions in the specification.* Changes in the specification may produce a difference in the sequences of requests generated by *RESTler*, which follows the ordering of the specification. For example, if a new request is added, comparing the sequences in order will produce many false positives, since the request sequences will be “out of sync”.
- *Non-determinism due to the global state of the service.* During exercising a service, many resources are created, which are asynchronously garbage collected in order to prevent exceeding resource limits. Due to timing differences between multiple runs, a `GET` request that, e.g., lists all blog posts, may return a different set of blog posts each time it is invoked (depending on how many have been deleted).

Although the above problems can be solved by further transformations of the precise diffs, our analysis of these diffs (see Section 4) motivated us to introduce a more compact format, focused on maximizing the relevant differences shown, to be able to more quickly discover and confirm a subset of regressions. We call this approach *stateless differing* of network logs, because each request-response pair is considered in isolation, without taking into account the full

```

PUT [...]/virtualNetworks/virtualNetworkName
- Request hash=abe64568e3c0b694c7413c6df808cf4c
+ Request hash=b2b7584635adcc0786cbeb240f4b867
  Responses:
- 201 Created
+ 400 Bad Request
-DELETE [...]/virtualNetworks/virtualNetworkName

```

**Figure 7: Stateless diff example.**

sequence of requests executed in the test run prior to the request. Specifically, stateless diffing transforms the network logs as follows:

- Group request-response pairs into equivalence classes, instead of the original ordered sequences.
- Consider only the response’s status, instead of the full content of its response.
- Do not show *added* requests and responses in the diff, since those are backwards-compatible feature additions of a specification and service.

Expanding on the first point, stateless diffing first groups together all requests of a particular type. A *request type* consists of the request’s HTTP method, the abstracted path, and a hash of its headers and body. As an example, a request like GET /blog/post-42 <headers...> <body...> has the following tuple as its type: GET, /blog/post-id (as described earlier, our abstractions replace the random id 42 with a constant), and *hash(headers||body)*. Each unique request type can correspond to many actual requests, because (1) the same request can be sent multiple times in different sequences, and (2) because abstractions map different concrete requests to the same abstracted one. Under each request type, we then list all the corresponding responses to those requests. For responses, to exclude any global state, we include only the response status code and description.

Figure 7 gives an example of a stateless diff. The first line states that a PUT request to the virtualNetworks API with the endpoint virtualNetworkName is present in both network logs (shown in black). The second and third line indicate that the full requests, including their bodies (shown by the hashes) changed, e.g., because a change in the specification added or removed properties in the request body. As a result, the server responds with 400 Bad Request after the update instead of 201 Created, a potential regression. Finally, in the last line, we also see that a DELETE request is missing in the network log after the update. In this case, the DELETE request is missing, because the resource to be deleted could not be created by the earlier PUT.

Stateful and stateless diffing will be experimentally evaluated in Section 4. In total, their implementation required about 2,200 lines of F# code.

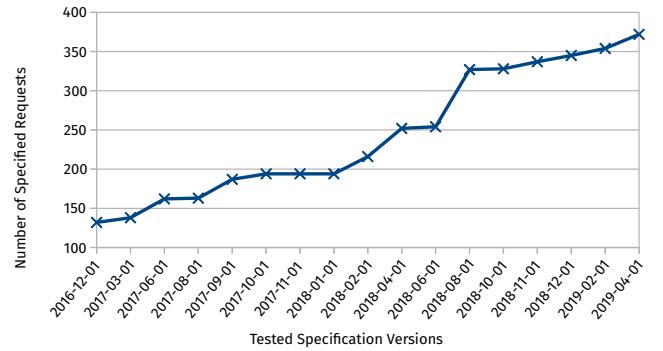
## 4 EVALUATION

### 4.1 Tested REST APIs and Versions

To evaluate our approach, we regression-tested REST APIs of widely used Microsoft Azure cloud services related to networking. They are used, for example, to allocate IP addresses and domain names, or to

**Table 1: Overview of the specifications of the tested APIs.**

Specifications in Directory	# Versions		Last Version		Sum Tested Versions	
	Total	Tested	# Files	# Lines	# Files	# Lines
dns/	4	3	60	4,142	129	9,585
network/	23	17	435	58,419	4,204	613,953
Sum			495	62,561	4,333	623,538

**Figure 8: Evolution of the Network APIs over time in terms of specified requests.**

provide higher level infrastructure, such as load balancers and firewalls.<sup>1</sup> Their Swagger specifications are available on GitHub [36].

Concretely, we looked at a collection of over 30 different REST APIs, split into two groups. First, the individual API of the Azure DNS service (in the GitHub repository under specification/dns/) and second, a large collection of APIs related to other networking services (jointly specified in specification/network/). We will refer to the first individual API as “DNS API” and to the large second group as “Network APIs” throughout the rest of the evaluation.

Since counting APIs is somewhat arbitrary, Table 1 gives an impression of the size of the tested APIs in more well-defined measures. The DNS API has 4 versions, where the most recent version is specified in 60 files containing 4,142 lines of JSON in total. The Network APIs are a substantially larger set, having 23 versions over time, with the latest version alone having a Swagger specification that spreads over 435 files with 58,419 lines of JSON. A large fraction of the specifications are examples of concrete requests (for the latest Network APIs: 395 files with 24,267 lines in total, so roughly 40% in terms of lines). Those examples are an important part of the specifications since (1) documentation is generated from them and (2) developers often use examples in their very first interaction with the service, so their correctness is important to service adoption.

Of all API versions, we selected those where the specification contained examples, which was the case for 3 versions (out of 4) for Azure DNS and 17 (out of 23) versions for the Network APIs. We selected those for two reasons: they are the most-recent ones (only the oldest versions do not include examples), and RESTler generates better test suites when given a set of examples to start from.

Many Azure APIs, including the ones we tested, are versioned by date (e.g., a version is 2019-04-01). A new version of the DNS and

<sup>1</sup><https://azure.microsoft.com/en-us/product-categories/networking/>

**Table 2: Overview of regressions found in Azure DNS and Network REST APIs.**

#	API Endpoint	Description	Correct vs. Regressed	Status
Regressions found with <i>stateless diffing</i> across different specification versions (“↑ direction”):				
1	publicIPAddress	Property in example incorrectly moved	2018-06-01	2018-07-01 Example corrected in 2018-08-01
2	publicIPAddress	Property in example incorrectly moved	2018-10-01	2018-11-01 “Re-regression” of bug above, corrected again in 2018-12-01
3	virtualNetworks	Property in example incorrectly moved	2018-10-01	2018-11-01 Example corrected in 2018-12-01
4	networkSecurityGroups	Required property in example removed	2018-10-01	2018-11-01 Example corrected in 2018-12-01
5	interfaceEndpoints	Whole endpoint removed / renamed	2019-02-01	2019-04-01 Confirmed, but deemed acceptable
Regressions found with <i>stateful diffing</i> across different service versions (“↔ direction”):				
6	dnsZones	Unspecified property zoneType in service response	2017-09-01	2017-10-01 Property specified in 2018-05-01
7	publicIPAddress	Unspecified property ipTags	2017-09-01	2017-10-01 Property specified in 2017-11-01
8	loadBalancers	Unspecified property enableDestinationServiceEndpoint	2017-10-01	2017-11-01 Property never specified
9	loadBalancers	Unspecified property allowBackendPortConflict	2018-10-01	2018-11-01 Property never specified
10	virtualNetworks	Unspecified property enableDdosProtection	2017-06-01	2017-08-01 Property specified in 2017-09-01
11	virtualNetworks	Unspecified property enableVmProtection	2017-06-01	2017-08-01 Property specified in 2017-09-01
12	virtualNetworks	Unspecified property delegations	2018-02-01	2018-04-01 Property specified in 2018-08-01
13	virtualNetworks	Unspecified property privateEndpointNetworkPolicies	2019-02-01	2019-04-01 Property specified in later commit
14	virtualNetworks	Unspecified property privateLinkServiceNetworkPolicies	2019-02-01	2019-04-01 Property specified in later commit

Network APIs is released approximately every two months (median interval: 61 days). Given this update frequency, it is crucial that new versions are backwards-compatible with previous versions, otherwise developers could not keep up with such frequent changes. Figure 8 shows the 17 versions we tested on the x-axis, from an early version 2016-12-01 up to the most recent version (at the time of testing) of 2019-04-01. It also shows the number of specified requests (i.e., HTTP method and endpoint) at each version, which grows from 132 to 372 over time. The large number of versions especially of the Network APIs lends itself well to historical analysis we perform in the following, i.e., finding regressions not only in the most recent update, but also in updates between earlier versions.

In total, we tested versions of REST APIs that sum up to more than 600,000 lines of Swagger specifications. This does not include the many more lines of code needed to implement these services, since we view the service implementations as black boxes. To keep up with these changes and find regressions in this amount of data, developers clearly need tooling, such as our approach.

## 4.2 Experimental Setup

To test the full upper-triangular API-version matrix as described in Section 2, we first download the mentioned specification versions. For each specification version, we then generate a client with *RESTler*. This client automatically sends requests conforming to said specification. Each generated client (one per specification version) is run multiple times, each time targeting a service of the same or a higher API version (by setting the *api-version* query parameter). Each test run (one combination of specification and service version) produces a network log. In total, testing  $N = 3$  versions of the DNS API results in  $T(N) = \frac{N(N+1)}{2} = 6$  network logs and testing 17 versions of the Network APIs results in 153 network logs.

For differential regression testing, we then diff each pair of network logs that correspond to adjacent cells in the client-server version matrix (i.e., where either the client or the service was updated by one version). This gives  $D(N) = N(N - 1) = 6$  diffs for

testing the DNS API and 272 diffs for the Network APIs. For service updates along the horizontal direction, we used the more precise stateful diffing, for updates of the underlying specification, we use stateless diffing. We substantiate this choice in Section 4.5. Manual inspection of the produced diffs resulted in 14 found regressions, which we discuss next.

## 4.3 Found Regressions – Overview

Table 2 gives an overview of the regressions we found in the tested DNS and Network APIs. In total, we discovered 14 unique regressions by differential testing. 5 were found by stateless diffing across different specification versions, i.e., when we updated the specification and generated a new client from it, but targeted the same service version. In these cases, the *Correct vs. Regressed* column shows the two versions of the specification update that introduced the regression. 9 regressions were found by stateful diffing across different service versions. In those cases, *Correct vs. Regressed* shows the service update, and not the specification, since the latter remains constant. The presence of both types of regressions shows that both directions of diffing in the version matrix are necessary. We also see from the *API Endpoint* column (the last path component of the URI) that we found regressions in different APIs and endpoints.

In the *Status* column, we show a short assessment of the bugs. In all but two cases (regressions #8 and #9), later versions of the API specification include fixes for the found regression, or other evidence (regression #5) is available to confirm the regression; these regressions were thus eventually found by the Azure service owners *after deployment*.

## 4.4 Found Regressions – Examples

We now discuss in more detail examples of regressions we found and why it is important to find them, ideally *before deployment*.

*Regressions #1 to #4: Broken Example Payloads.* Errors in examples – from which documentation is generated and that illustrate

```

1 | { "parameters": {
2 |   ...
3 |   "publicIpAddressName": "test-ip",
4 |   "parameters": {
5 |     "location": "eastus"
6 |   }
7 | }
8 |
9 | }
10}

```

Correct versions: 2018-06-01  
and 2018-10-01.

```

1 | { "parameters": {
2 |   ...
3 |   "publicIpAddressName": "test-ip",
4 |   "parameters": {}, "location": "eastus"
5 | }
6 | }
7 |
8 |
9 |
10}

```

Regressed version 2018-07-01 and  
re-regressed version 2018-11-01.

**Figure 9: Regression in the examples of the specification for publicIpAddress. The erroneous change is highlighted.**

how to use an API – are frustrating for users because they are typically the first attempt to use the service. In specification version 2018-07-01, a specification author erroneously moved the location property one level up in an example request for the publicIpAddress API (regression #1), as shown in Figure 9. This was caught by stateless diffing, because both network logs contain the request, but the response was different: 200 OK in the old version, but 400 Bad Request in the regressed version. In the next specification version, 2018-08-01, the change was reverted and the bug fixed. Interestingly though, the example was regressed again in version 2018-11-01 (regression #2). We can see how the nesting of two parameters properties might have led to the confusion. From the double regression we can also take that tooling such as described in this paper would help developers catch specification regressions prior to release. Even more interestingly, our tool also found similar regressions in examples for other APIs (regressions #3 and #4).

*Regression #5: Renamed Endpoint.* A renamed or removed endpoint in the specification is a severe regression, because existing clients that use it will break when updating to the next service version. In this case, the stateless diff of two network logs showed:

`-GET [...]/providers/Microsoft.Network/interfaceEndpoints`

That is, the request to this endpoint was previously sent by the client generated from the old specification, but no longer after the specification update. When investigating the bug, we found that actual users had already detected such breaks and opened a GitHub issue. The response of the service owners was that this breaking change was intentional, and acceptable to them because the API was not yet officially announced. This nevertheless appears to be a regression, particularly because the previously working version did not end with “*-preview*”, a suffix typically used to differentiate such new unstable APIs that may change in the future.

*Regressions #6 to #14: Unspecified Properties in Responses.* An additional, but unspecified JSON property in the response of an API may break clients, because the property cannot be parsed by a generated SDK or is not handled by custom code written by the user based on the specification. Microsoft’s REST API Guidelines [24] (Sec 12.3) mention this explicitly: “Azure defines the addition of a new JSON field in a response to be not backwards compatible”. For instance, a property named ipTags started to be returned in responses by service version 2017-10-01 (regression #7). However, this property

**Table 3: Size comparison of stateful vs. stateless diffs for DNS and Network APIs depending on the update direction.**

Diffing Method	Update Direction	Sum Lines	Sum Bytes	Empty Diffs (of 139 in total)
stateful	specification	8,784,417	677,605,439	14 (10%)
stateful	service	388,757	34,951,726	0
stateless	specification	2,197	505,298	56 (40%)
stateless	service	95	29,394	136 (98%)

**Table 4: Diff sizes with the diagonal-only strategy.**

Diffing Method	Update Direction	Sum Lines (Reduction)	Sum Bytes (Reduction)	Found Bugs
stateful	service	36,637 (9%)	3,690,016 (11%)	9 (of 9)
stateless	specification	302 (14%)	63,774 (13%)	5 (of 5)

was added in the specification only a month later at version 2017-11-01, thus breaking clients in the meantime. Perhaps the service owners intended to expose these properties to clients, but did so “too early”, namely in service versions where the specifications do not yet mention these new properties. Our approach found nine of such cases, because the stateful diff showed those properties as inserts in the received response bodies.

## 4.5 Quantitative Evaluation

We now evaluate individual parts of our approach in more detail.

*4.5.1 Comparing Stateful vs. Stateless Diffing.* As expected, stateful diffs contain more information on average (2.7 MB per textual diff file) than stateless diffs (6 KB). Indeed, stateless diffs only consider request types (i.e., HTTP methods and endpoints) and the response status, but ignore request and response bodies.

Table 3 shows that the sizes of the diffs also depend on the direction of the comparison, i.e., whether the specification or the service version remains constant when diffing a pair of network logs. Stateful diffing across specification updates produces a lot of differences, in total 678 MB of text files. The root cause for these large diffs is that new requests or parameters in the specification will appear in the network log of the new client, and add up quickly if the request is tested many times. Since this is a lot of data to manually inspect, we do not make use of this output in our approach (and show it grayed-out in the table). In contrast, stateless diffing across specification updates (third row), produces a much more manageable amount of diffs to inspect, namely 505 KB or about two thousand lines in total.

For service updates (i.e., where the compared network logs come from the same client derived from a constant specification version), stateless diffing abstracts away too much information and brings up almost no warnings (last row). 98% of the diffs in that case are empty. When diffing network logs from the same client, we thus use stateful diffing (second row), which is both feasible in terms of output produced and more precise, because the diff may contain changes in the services response bodies (e.g., added properties).

**Table 5: Relation of regressions to diffs in which they manifest and if the diagonal-only diffing strategy finds them.**

Regression:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
In # Diffs	7	4	4	4	1	1	5	6	13	3	3	9	12	12
On Diagonal?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

**4.5.2 Upper-Triangular Matrix vs. Diagonal-Only Strategy.** For a comprehensive evaluation, we inspected all 278 diffs (DNS: 6, Network APIs: 272) of the upper-triangular matrix of specification and service versions, and found the 14 regressions from Table 2. In Section 2.6 we additionally propose a diagonal-only strategy that requires inspection of only two updates (and thus diffs) per new API version: First, diffing the network logs of  $(c_n, s_n)$  against  $(c_n, s_{n+1})$  when the service is updated, and then diffing  $(c_n, s_{n+1})$  against  $(c_{n+1}, s_{n+1})$  when the new specification is published. This reduces the number of diffs greatly – linear instead of quadratic in terms of API versions. With the tested APIs, the diagonal-only subset of diffs comprises only 36 files (DNS: 4, Network APIs: 32) instead of 278. Table 4 shows that this results in a large reduction in terms of lines and bytes: stateful diffing across specification versions produces only 36,637 lines to inspect instead of 388,757, i.e., a reduction to 9% of the original effort. For stateless diffing across specification versions, there are only 302 lines instead of 2,197 to inspect, or a reduction to 14%.

Table 5 shows the mapping of regressions to diffs, i.e., in how many diffs each regression appears. For instance, regression #9 appears in 13 distinct diff files. We can see that despite this reduction in inspection effort, the diagonal-only strategy still manages to find *all 14 regressions* in total. This strategy thus provides a more *attractive diffs-to-regression ratio* than the upper-triangular-matrix strategy, especially for large numbers  $N$  of API versions.

**4.5.3 Runtime and Manual Inspection Effort.** All experiments were conducted on a single commodity PC with an Intel Core i7 processor and 32GB of main memory under Windows 10. For the DNS API, collecting the network logs for all valid combinations of specification and service versions took 11 minutes and 28 seconds. For the much larger Network APIs, this step took 3 days, 8 hours, and 4 minutes. During this interaction with the Azure services, a total of 92,381 requests were sent by RESTler. Running our diffing tools for all network logs took less than an hour for all valid pairs of logs in total. That is, the runtime of our approach is clearly dominated by exercising the web services, not the diffing of the collected logs.

Analyzing all 278 diffs of all the tested APIs for this historical analysis took about a week of work for a PhD student. In particular, analyzing stateless diffs was relatively quick since 40% were empty and the remaining ones were often duplicates of each other. Since stateful diffs contain more information (e.g., bodies of requests and responses), inspecting them took more time but was still manageable since differences, e.g., in the response of a particular request, re-appear several times and can be skipped. For this, we made use of code folding for JSON files (which stateful diffs are).

## 4.6 Threats to Validity

Even though we evaluate our approach on a large collection of real-world REST APIs, some threats to validity remain.

Our approach uses *RESTler* to automatically generate requests from an API specification, and test the service with those sequences. The amount of regressions that can be uncovered thus depends on how well *RESTler* can exercise a service under test. However, our approach makes no assumption on how requests are generated. The method for generating requests is a black box that can be exchanged or improved independently of our core differential testing idea.

The results of our evaluation also depend on the effectiveness of our diffing described in Section 3. Future work could improve our implementation by using more elaborate diffing algorithms and thus reduce the number of false positives. The core idea of our approach, namely the difference between specification and service regressions and that testing for both is essential, remains valid.

Finally, our experimental results also depend on the specific APIs considered, as well as the bugs in those APIs and services. While the specifications of the Azure networking APIs live in a single repository, this is merely an artifact of centralizing these specifications for developer convenience. These APIs are distinct, targeting different services, and written and maintained by different service owners.

## 4.7 Continuous Differential Regression Testing

We presented a *historical analysis* of Azure networking APIs in order to *evaluate* differential regression testing on a large amount of data, to see how many changes (diffs) there are, how many regressions we can find using our approach, what these regressions look like, and to determine which differential testing strategy (upper-triangular matrix or diagonal-only) works best.

However, the main practical use of our approach is *continuous differential regression testing*, whereby differential testing is applied to detect regressions in the *latest-only* API version update *before deployment*. Another similar main application is the pre-deployment detection of service regressions across (daily or weekly) service *revisions*, i.e., service updates which do *not* involve a new API version. In this scenario, the latest API specification is fixed and the testing results obtained with today’s service are compared only with those obtained during the last testing results available.

In such a *continuous testing* context, there are *at most two diff* files to inspect, and their sizes is typically *much smaller* than when comparing very different versions as in our historical analysis. In particular, they are empty when adding a new endpoint, adding new examples, or updating description text. Also, within a continuous-integration environment, API changes are already reviewed since they may break clients (e.g., see [35]). Our automation provides a systematic way for API changes to be presented to developers. Inspecting a small diff file only requires minutes by API experts.

Yet regressions can still be found this way. As an example, we found a regression across a revision of the DNS service version 2017-10-01 between May 5, 2019 and May 19, 2019 with respect to the unspecified property `zoneType` (see regression #6 in Table 2). A manual analysis revealed that, on May 5, all versions of the service returned the extra property, even though it was only declared in the newer specification of 2018-05-01. Then, on May 19, it appears that

a fix was attempted: the property was removed from the (dynamic) response of several of the earlier service versions, matching the corresponding specifications, except for service version 2017-10-01, where the specification and service were still out-of-sync.

## 5 RELATED WORK

*REST APIs and Their Versioning.* Since the REST principle was introduced in [21, 22], REST APIs have become a building block of the modern web, and a foundation of cloud services. REST API versioning is discussed in books like [7, 33, 40], where backwards compatibility is emphasized as a golden-rule of good REST API design and evolution. However, we are not aware of any other work discussing the two types of regressions, namely service vs. specification regressions, defined in Section 2 and how to detect these with differential testing. This paper introduces a *principled foundation* to this important topic.

*Regression Testing for REST APIs.* Regression testing [39], i.e., running a test suite after modifying software in order to ensure that existing features are still working correctly, is a widely-adopted form of testing [29], including in industry and practice [41]. With the recent explosion of web and cloud services, regression testing has naturally been applied to REST APIs as well, mostly in commercial tools, such as SoapUI [5], Postman [3], and others [2, 4]. These tools are helpful to develop a fixed regression test suite, possibly by first recording live or manually-generated test traffic. However, these tools (1) do not generate tests automatically and they (2) do not perform differential testing.

*Test Generation Tools for REST APIs.* Other tools available for testing REST APIs generate *new tests* by capturing API traffic, and then parsing, fuzzing and replaying the new traffic with the hope of finding bugs [9, 10, 14, 15, 43, 45, 47]. Other extensions are possible if the service code can be instrumented [11] or if a functional specification is available [44]. *RESTler* [12] is a recent tool that performs a lightweight static analysis of a Swagger specification in order to infer dependencies among request types, and then automatically generates *sequences* of requests (instead of single requests) in order to exercise the service behind the API more deeply, in a *stateful manner*, and *without pre-recorded API traffic*. These tools can find bugs like 500 Internal Server Errors, but none of these perform differential testing or target regressions specifically.

*Differential Testing.* Originally devised by McKeeman [34], differential testing is now a well-known technique for solving the oracle problem [13] in automated testing by comparing two different, but related programs on the same inputs. One way of comparing different programs is by taking different implementations, e.g., for compiler testing [16, 18, 27, 31, 48]. Since then, differential testing has also been applied to other development tools, such as IDEs [17], interactive debuggers [28], and program analyzers [26]. In this work, we also use differential testing, but apply it to REST APIs in order to compare various client-service configurations.

*Static Differing to Find Specification Regressions.* Our stateful and stateless approaches to differing network logs are built upon standard algorithms for computing edit distances and edits scripts [38]. Differing can be applied directly to Swagger specifications in order

to find some types of clear regressions, such as removing a request from an API [1]. However, static differencing of specifications has its own challenges. First, these diffs can be large: for instance, the 16 diff files obtained by pairwise differing all 17 Azure Network API Swagger specifications in chronologic order result in a total of 15,571 lines and 465,624 bytes. Second, most differences are harmless: adding new files, reorganizing API requests, adding or modifying examples, etc. Third, even when modifying specific parameters in a new specification version, the old parameters are often still handled correctly by the new service for backwards-compatibility reasons, and therefore are *not* regressions; the only way to eliminate such false alarms is by *dynamically* testing the new service against (client code generated from) the old specification. In contrast, specification regressions can be found with stateless differing and the diagonal-only strategy (16 diffs) by inspecting only 302 lines (see Table 4), i.e., two orders of magnitude less data compared to 15,571 lines with static differing. Overall, static specification differing is complementary to the dynamic testing-based approach developed in this work.

## 6 CONCLUSION

This paper introduces differential regression testing for REST APIs. It is the first to point out the key distinction between *service and specification regressions*. We discussed how to detect such regressions by comparing network logs capturing REST API traffic using stateful and stateless differing. To demonstrate the effectiveness of this approach on a large set of services and APIs, we presented a detailed API history analysis of Microsoft Azure networking APIs, where we detected 14 regressions across 17 versions of these APIs. We discussed how these regressions were later fixed in subsequent specification versions and service deployments.

The main application of our approach is continuous differential regression testing, whereby differential testing is applied to detect regressions in the latest API version of a service *before* its deployment, hence avoiding expensive regressions affecting customers. We plan to deploy our tools widely soon, and we hope the evidence provided in this paper will encourage adoption.

## ACKNOWLEDGMENTS

We thank Albert Greenberg, Anton Evseev, Mikhail Triakhov, and Natalia Varava from the Microsoft Azure Networking team for encouraging us to pursue this line of work and for their comments on the results of Section 4.

## REFERENCES

- [1] [n.d.] 42crunch. <https://42crunch.com/>
- [2] [n.d.] Apigee Docs. <https://docs.apigee.com/>
- [3] [n.d.] Postman / API Development Environment. <https://www.getpostman.com/>
- [4] [n.d.] vREST – Automated REST API Testing Tool. <https://vrest.io/>
- [5] [n.d.] The World’s Most Popular Testing Tool / SoapUI. <https://www.soapui.org/>
- [6] 2019. Azure SDK. <https://github.com/Azure/azure-sdk>
- [7] S. Allamaraju. 2010. RESTful Web Services Cookbook. O’Reilly.
- [8] Amazon. 2019. Amazon Web Services (AWS) - Cloud Computing Services. <https://aws.amazon.com/>
- [9] APIFuzzer [n.d.] APIFuzzer. <https://github.com/KissPeter/APIFuzzer>.
- [10] AppSpider [n.d.] AppSpider. <https://www.rapid7.com/products/appspider>.
- [11] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology* 28, 1 (2019).
- [12] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) (ICSE ’19). IEEE Press, Piscataway, NJ, USA, 748–758. <https://doi.org/10.1109/ICSE.2019.00083>

- [13] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.
- [14] Boofuzz [n.d.]. BooFuzz. <https://github.com/jtpereyda/boofuzz>.
- [15] Burp [n.d.]. Burp Suite. <https://portswigger.net/burp>.
- [16] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 197–208.
- [17] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) (ESEC/FSE '07). ACM, New York, NY, USA, 185–194. <https://doi.org/10.1145/1287624.1287651>
- [18] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. <https://doi.org/10.1145/3133917>
- [19] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. 2014. Web API growing pains: Stories from client developers and their code. In *Proceedings of the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)* (Antwerp, Belgium).
- [20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Västerås, Sweden) (ASE '14). Association for Computing Machinery, New York, NY, USA, 313–324. <https://doi.org/10.1145/2642937.2642982>
- [21] Roy T. Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation.
- [22] Roy T. Fielding and Richard N. Taylor. 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)* 2, 2 (2002), 115–150.
- [23] Zack Grossbart. 2019. JSON Diff – The semantic JSON compare tool. <http://www.jsondiff.com/>
- [24] Microsoft REST API Guidelines Working Group. 2019. *Microsoft REST API Guidelines*. <https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md>
- [25] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashm, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow?. In *Proceedings of the 2013 21st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Saint Petersburg, Russia) (ESEC/FSE 2013). ACM, New York, NY, USA.
- [26] Christian Klinger, Maria Christakis, and Valentin Wüstholtz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 239–250.
- [27] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) (PLDI '14). ACM, New York, NY, USA, 216–226. <https://doi.org/10.1145/2594291.2594334>
- [28] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). ACM, New York, NY, USA, 610–620. <https://doi.org/10.1145/3236024.3236037>
- [29] Hareton KN Leung and Lee White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*. IEEE, 60–69.
- [30] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. 2013. How Does Web Service API Evolution Affect Clients?. In *Proceedings of the 2013 IEEE 20th International Conference on Web Services* (Santa Clara, CA).
- [31] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) (PLDI '15). ACM, New York, NY, USA, 65–76. <https://doi.org/10.1145/2737924.2737986>
- [32] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of HotOS 2019* (Bertinoro, Italy).
- [33] Mark Masse. 2011. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.".
- [34] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [35] Microsoft. 2019. *Azure ARM API Review Checklist*. <https://github.com/Azure/azure-rest-api-specs/pull/6632>
- [36] Microsoft. 2019. *Azure REST API Specifications*. <https://github.com/Azure/azure-rest-api-specs>
- [37] Microsoft. 2019. *Microsoft Azure Cloud Computing Platform & Services*. <https://azure.microsoft.com/en-us/>
- [38] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1 (01 Nov 1986), 251–266. <https://doi.org/10.1007/BF01840446>
- [39] G. J. Myers. 1979. *The Art of Software Testing*. Wiley.
- [40] S. Newman. 2013. *Building Microservices*. O'Reilly.
- [41] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression Testing in an Industrial Environment. *Commun. ACM* 41, 5 (May 1998), 81–86. <https://doi.org/10.1145/274946.274960>
- [42] Tom Preston-Werner. 2019. *Semantic Versioning 2.0.0*. <https://semver.org/>
- [43] QualysWAS [n.d.]. Qualys Web Application Scanning (WAS). <https://www.qualys.com/apps/web-app-scanning/>
- [44] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. *ACM Transactions on Software Engineering* 44, 11 (2018).
- [45] Sulley [n.d.]. Sulley. <https://github.com/OpenRCE/sulley>.
- [46] Swagger [n.d.]. Swagger. <https://swagger.io/>
- [47] TnT-Fuzzer [n.d.]. TnT-Fuzzer. <https://github.com/Teebytes/TnT-Fuzzer>.
- [48] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) (PLDI '11). ACM, New York, NY, USA, 283–294. <https://doi.org/10.1145/1993498.1993532>

# Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization

Qianyang Peng  
University of Illinois  
Urbana, IL, USA  
qp3@illinois.edu

August Shi  
University of Illinois  
Urbana, IL, USA  
awshi2@illinois.edu

Lingming Zhang  
University of Texas at Dallas  
Dallas, TX, USA  
lingming.zhang@utdallas.edu

## ABSTRACT

Test-case prioritization (TCP) aims to detect regression bugs faster via reordering the tests run. While TCP has been studied for over 20 years, it was almost always evaluated using seeded faults/mutants as opposed to using *real test failures*. In this work, we study the recent change-aware information retrieval (IR) technique for TCP. Prior work has shown it performing better than traditional coverage-based TCP techniques, but it was only evaluated on a small-scale dataset with a cost-unaware metric based on seeded faults/mutants. We extend the prior work by conducting a much larger and more realistic evaluation as well as proposing enhancements that substantially improve the performance. In particular, we evaluate the original technique on a large-scale, real-world software-evolution dataset with real failures using both cost-aware and cost-unaware metrics under various configurations. Also, we design and evaluate hybrid techniques combining the IR features, historical test execution time, and test failure frequencies. Our results show that the change-aware IR technique outperforms state-of-the-art coverage-based techniques in this real-world setting, and our hybrid techniques improve even further upon the original IR technique. Moreover, we show that flaky tests have a substantial impact on evaluating the change-aware TCP techniques based on real test failures.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Test-case prioritization, information retrieval, continuous integration

### ACM Reference Format:

Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397383>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397383>

## 1 INTRODUCTION

Test-case prioritization (TCP) aims to detect regression bugs faster by reordering the tests such that the ones more likely to fail (and therefore detect bugs) are run first [50]. To date, researchers have proposed a large number of TCP techniques, including both *change-unaware* and *change-aware* techniques. Change-unaware techniques use dynamic or static information from the old version to perform TCP. For example, the *total* technique simply sorts all the tests in the descending order of the number of the covered program elements (e.g., methods or statements), while the improved *additional* technique sorts the tests based on the number of their covered elements that are not covered by already prioritized tests [51]. In contrast, change-aware techniques consider program changes between the old and new software versions to prioritize tests more likely to detect bugs due to these changes. For example, a change-aware technique based on information retrieval (IR) [53] reduces the problem of TCP into the traditional IR problem [37]—the program changes between revisions are treated as the *query*, while the tests are treated as the *data objects*; the tests that are textually more related to the program changes are prioritized earlier.

Although various studies have investigated the effectiveness of existing TCP techniques, they suffer from the following limitations. First, they are usually performed on artificial software evolution with seeded artificial or real bugs, and not on real-world software evolution with *real test failures*. For example, recently Luo *et al.* [34] empirically evaluated TCP techniques on both artificial bugs via mutation testing and real bugs from Defects4J [22]. However, even Defects4J bugs are not representative of real regression bugs, because Defects4J bugs are isolated bugs, while real regression bugs often come with other benign changes. Furthermore, developers using TCP as part of their workflow would not know *a priori* what are the exact bugs in the code; they would only be able to observe the test failures that occur. Second, the existing TCP techniques are usually evaluated using *cost-unaware metrics*, such as Average Percentage of Faults Detected (APFD) [50]. Even state-of-the-art TCP techniques were also evaluated using only APFD [33, 44, 53], which can be biased and unrealistic [10, 36]. Last but not least, *flaky tests* have been demonstrated to be prevalent in practice [32], but to the best of our knowledge, none of the existing studies of TCP considered the impacts of flaky tests. In evaluating TCP effectiveness, flaky test failures would not indicate real bugs in the code, so having such failures can mislead the results if the goal is to order the tests that fail (because presumably they detect a real bug) first.

In this work, we focus on the IR technique for TCP based on program changes [53], which we call the *IR-based TCP technique*. This technique uses IR to calculate the textual similarity between the program changes and each test, then prioritizes the tests in

descending order of their similarities with the changes. We focus on this specific technique because it has been shown to outperform traditional change-unaware total and additional techniques [53]. Furthermore, the IR-based TCP technique is a lightweight static technique that scales well and can be easily applied to a large set of projects. In our new evaluation, we aim to reduce the limitations of the prior studies by using a dataset consisting of real test failures in projects that occur due to real software evolution and by using both cost-unaware and cost-aware metrics for evaluation. We also aim to further enhance the IR-based TCP technique by exploring more configurations and hybridizing IR features with historical test execution time and failure frequencies to improve its performance.

Some other studies have started using real test failures for other aspects of regression testing, e.g., for test-suite reduction [56] regression test selection (RTS) [7], and the impact of flaky tests [25]. Elbaum *et al.* proposed their TCP and RTS techniques and evaluated them on a closed-source Google dataset [14]. In contrast, we construct a dataset using real-world, open-source projects.

This paper makes the following contributions:

- *Dataset*: We collect a large-scale dataset from GitHub and Travis CI. Our dataset consists of 2,042 Travis CI builds with 2,980 jobs (a Travis CI build can have multiple jobs), and 6,618 real test failures from 123 open-source Java projects. For each job, our dataset has the source code, program changes, and the test execution information (the pass/fail outcome and the test execution time), which suffice for evaluating TCP techniques. Our dataset also includes the test coverage information for 565 jobs and the test flakiness labeling for 252 jobs. Moreover, our dataset is publicly available [4].
- *Novel Technique*: We propose new hybrid IR-based TCP techniques that takes into consideration not only change-aware IR but also test execution time and historical test failure frequencies. Our hybrid IR-based TCP techniques significantly outperform the traditional IR-based TCP techniques and history-based techniques.
- *Evaluation*: We evaluate a wide range of configurations of IR-based TCP techniques on our dataset using both cost-unaware and cost-aware metrics. We are among the first to evaluate TCP techniques based on a large number of real test failures taken from real-world project changes as opposed to seeded mutants and faults in the code. We are the first to evaluate specifically IR-based TCP techniques with a cost-aware metric, and we are the first to study the impacts of flaky tests on TCP, which pose a problem to evaluations based on test failures.
- *Outcomes*: Our study derives various outcomes, including: (1) there is a huge bias when using a cost-unaware metric for TCP instead of a cost-aware metric; (2) the IR-based TCP technique outperforms state-of-the-art coverage-based TCP techniques on real software evolution by both cost-unaware and cost-aware metrics; (3) our hybrid IR-based TCP techniques are very effective in performing TCP and significantly outperform all traditional IR-based TCP techniques and history-based techniques; and (4) flaky tests have a substantial impact on the change-aware TCP techniques.

## 2 INFORMATION RETRIEVAL (IR) TECHNIQUES

Saha *et al.* [53] first proposed using information retrieval (IR) techniques to perform test-case prioritization (TCP). The key idea is that tests with a higher textual similarity to the modified code are more likely related to the program changes, thus having a higher chance to reveal any bugs between the versions due to the changes.

In general, in an IR system, a *query* is performed on a set of *data objects*, and the system returns a ranking of the data objects based on similarity against the input query. There are three key components for an IR system: (1) how to construct the data objects, (2) how to construct the query, and (3) the retrieval model that matches the query against the data objects to return a ranking of the data objects as the result. We describe each component in detail and how they relate to performing TCP.

### 2.1 Construction of Data Objects

For IR-based TCP, the data objects are constructed from the tests<sup>1</sup>. To construct the data objects, the first step is to process the string representation of each test into tokens. A *token* is a sequence of characters that act as a semantic unit for processing. For example, a variable can be a token. We consider four different approaches to process the test files.

The first and most naive approach we consider is to use the whitespace tokenizer that breaks text into tokens separated by any whitespace character. We denote this approach as *Low*. The approach is simple and straightforward, but it results in two main disadvantages: (1) the approach does not filter out meaningless terms for IR such as Java keywords (e.g., *if*, *else*, *return*), operators, numbers and open-source licenses; and (2) the approach fails to detect the similarities between the variable names that are partially but not exactly the same, e.g., *setWeight* and *getWeight*.

An optimization for the naive *Low* approach is to use a special code tokenizer to tokenize the source code; we denote this second approach as *Low<sub>token</sub>*. *Low<sub>token</sub>* improves upon *Low* by introducing a code tokenizer that (1) filters out all the numbers and operators, (2) segments the long variables by non-alphabetical characters in-between and by the camel-case heuristics, and (3) turn all upper-case letters to lower-case letters. With this code tokenizer, *Low<sub>token</sub>* removes several of the disadvantages of *Low*. However, *Low<sub>token</sub>* still does not filter out some meaningless terms for IR, such as Java keywords and open-source licenses.

A third approach is to build an abstract syntax tree (AST) from each test file and extract the identifiers and comments [53]. This step removes most meaningless terms. We denote this approach as *High*. After extracting the identifiers, we can additionally apply the same code tokenizer as *Low<sub>token</sub>* to create finer-grained tokens. We denote this fourth, final approach as *High<sub>token</sub>*.

### 2.2 Construction of Queries

IR-based TCP constructs a query based on the program changes (in terms of changed lines) between two versions. Similar to the construction of data objects, we also apply one of *Low*, *Low<sub>token</sub>*, *High*, and *High<sub>token</sub>* as the preprocessing approach for the query.

<sup>1</sup>In this work, we prioritize test classes as tests; a test class can contain multiple test methods. In the rest of this paper, when we refer to a test, we mean test class.

For each type of data-object construction, we apply the same corresponding preprocessing approach, e.g., if using  $High_{token}$  for data objects, we use  $High_{token}$  for the query as well.

#### Example 1: The context of changed code

```

1  private void initEnvType() {
2      // Get environment from system property
3      m_env = System.getProperty("env");
4      + if (Utils.isBlank(m_env)) {
5          +     m_env = System.getProperty("active");
6      }
7      if (!Utils.isBlank(m_env)) {
8          m_env = m_env.trim();
9          logger.info(info, m_env);
10         return;
11     }

```

An important consideration for constructing the query is whether to include the *context* of the change or not. By context, we mean the lines of code around the exact changed lines between the two versions. Example 1 illustrates a small code diff with the surrounding context, taken from a change to file `DefaultServiceProvider.java` in GitHub project `ctripcorp/apollo`<sup>2</sup>. In the example, lines 4-6 are the modified lines (indicated by +), and the highlighted lines show the considered context for the diff if configured to include 1 line of context. More context makes the query more informative, and the query has a higher chance to reveal hidden similarities between the code change and tests. However, more context also has a higher chance to include unrelated text into the query. Prior work [53] used 0 lines of context to construct queries without evaluating the impact of this choice of context. In our evaluation, we evaluate the impact of including 0 lines, 1 line, 3 lines, 5 lines of context before and after the changed lines, and we also consider including the contents of the whole file as the context.

### 2.3 Retrieval Models

The retrieval model part of the IR system takes the data objects and the query as input, and it generates a ranking of data objects (i.e., tests for IR-based TCP) as the output. In our evaluation, we explore the following four retrieval models: Tf-idf, BM25, LSI and LDA.

**Tf-idf.** Tf-idf [54] is a bag-of-words based text vectorization algorithm. Given the vector representations of the data objects, we perform the data-object ranking based on the vector distances between the data objects' vectors and the query. We use TfidfVectorizer from scikit-learn [45] to implement the Tf-idf model. We perform TCP by sorting the tests in a descending order of the cosine similarity scores between their Tf-idf vectors and the query's vector.

**BM25.** BM25 [48] (also known as Okapi BM25), is another successful retrieval model [47]. Unlike the Tf-idf model, BM25 takes the data-object lengths into consideration, such that shorter data objects are given higher rankings. Moreover, unlike the traditional Tf-idf model, which requires using the cosine similarity scores to perform the ranking after feature extraction, BM25 itself is designed to be a ranking algorithm that can directly compute the similarity scores. In our evaluation, we use the gensim [46] implementation of BM25, and we use their default parameters. We perform TCP by

<sup>2</sup><https://github.com/ctripcorp/apollo/pull/1029/files>

sorting tests in the descending order of their BM25 scores. Saha *et al.* [53] previously evaluated IR-based TCP using BM25 on Java projects, and recently Mattis and Hirschfeld also evaluated IR-based TCP using BM25, but on Python projects.

**LSI and LDA.** LSI [11] and LDA [62] are two classic unsupervised bag-of-words topic models. As a topic model, the representation of a data object or a query is a vector of topics rather than a vector of raw tokens. This mathematical embedding model transforms the data objects from a very high dimensional vector space with one dimension per word into a vector space with a much lower dimension. Using a vector of topics to represent each data object and the query, we can calculate the similarity scores and rank the data objects. We use the gensim implementation of LSI and LDA, and we use cosine similarity to calculate vector similarities to compute the ranking.

**2.3.1 New Tests and Tie-Breaking.** In the case of new tests, which means there is no IR score, we configure IR-based TCP to place the new tests at the very beginning of the test-execution order, because we believe new tests are the most related to the recent changes.

When multiple tests share the same score, it means that IR-based TCP considers them equally good. However, the tests still need to be ordered. In such a scenario, we order these tests deterministically in the order they would have been scheduled to run when there is no TCP technique that reorders them (basically running them in the order the test runner runs them).

## 3 CONSTRUCTING DATASET

To perform IR-based TCP, we need the code changes between two software versions (for the query) and the textual contents of tests (for the data objects). For our evaluation, we need test execution outcomes (pass or fail) and test execution time as to evaluate how well an IR-based TCP technique orders failing tests first. We construct our dataset that contains all of the above requirements.

To construct our dataset, we use projects that use the Maven build system, are publicly hosted on GitHub, and use Travis CI. Maven [2] is a popular build system for Java projects, and the logs from Maven builds are detailed enough to contain the information we need concerning test execution outcomes and times. GitHub [1] is the most popular platform for open-source projects. If a project is active on GitHub, we can download any historical version of the project, and we can get the code changes between any two code versions in the form of an unified diff file. Travis CI [3] is a widely-used continuous-integration service that integrates well with GitHub. When a project on GitHub is integrated with Travis CI, every time a developer pushes or a pull request is made to the GitHub project, that push or pull request triggers a Travis *build*, which checks out the relevant version of the project onto a server in the cloud and then proceeds to compile and test that version.

Each build on Travis CI can consist of multiple *jobs*. Different jobs for the same build run on the same code version, but can run different build commands or run using different environment variables. Each job outputs a different job log and has its own job state (passed, failed, or errored). As such, we treat jobs as the data points for running TCP on in our study, as opposed to builds. We use the Travis CI API to obtain the state of each job as well as the state of the jobs from the previous build. The Travis CI API also

provides for us the Git commit SHA corresponding to the build and the previous build as well. We use the commit SHAs to in turn query the GitHub API to obtain the diff between the two versions. Furthermore, we use the Travis CI API to download full job logs corresponding to each job. We obtain the test execution outcomes and test execution times for each job by parsing the historical job logs, as long as the job log is well formatted.

We start with 10,000 Java projects collected using GitHub's search API. A single query to the GitHub search API returns at most 1,000 projects, so we use 10 separate sub-queries by the year of creation of the project. In the search query, we specify the primary language to be Java and sort the result by the descending number of stars (a representation of the popularity of a project). Out of the 10,000 projects, we find 1,147 projects that build using Maven and also use Travis CI. For each of these projects, we select the jobs for the project that satisfy the requirements below:

- (1) The SHA of the build's corresponding commit is accessible from GitHub, so we can download a snapshot of the project at that commit in the form of a zip file.
- (2) The state of the job is “failed” due to test failures, while the state of the previous Travis build is “passed” (A “passed” Travis build means the states of all jobs in it are “passed”). This requirement is to increase the likelihood that the job contains regressions that are due to the code changes.
- (3) The logs of both the job and the previous job are analyzable, which means we can get the fully-qualified name (FQN), the outcome (passed or failed), and the execution time of every test run in the jobs.
- (4) The code changes between the build and the previous build contains at least one Java file, and the Java files are decodeable and parseable.

For each job log, we extract the FQNs and execution times of executed tests by matching the text with regular expressions “Running: (\*.?)” and “Time elapsed: (\*.? s)”. We extract the failed test FQNs with the provided toolset from *TravisTorrent* [9].

Given these requirements for jobs, we obtain a final dataset that consists of 2,980 jobs from 2,042 Travis builds across 123 projects. More details on the specific projects and the jobs we analyze can be found in our released dataset [4].

## 4 EXPERIMENT SETUP

We evaluate IR-based TCP on our collected dataset of projects and test failures. In this section, we discuss the metrics we use to evaluate TCP and give details on the other TCP techniques we compare IR-based TCP against.

### 4.1 Evaluation Metrics

We use two metrics, Average Percentage of Faults Detected (APFD) and Average Percentage of Fault Detected per Cost (APFDc), to evaluate TCP effectiveness.

**4.1.1 Average Percentage of Faults Detected (APFD).** Average Percentage of Faults Detected (APFD) [50] is a widely used cost-unaware metric to measure TCP effectiveness:

$$APFD = 1 - \frac{\sum_{i=1}^m TF_i}{n \times m} + \frac{1}{2n}.$$

In the formula,  $n$  is the number of tests and  $m$  is the number of faults.  $TF_i$  is the position of the first failed test that detects the  $i$ th fault in the prioritized test suite.

**4.1.2 Average Percentage of Fault Detected per Cost (APFDc).** Average Percentage of Fault Detected per Cost (APFDc) is a variant of APFD that takes into consideration the different fault severities and the costs of test executions [12, 36]. Due to the difficulty of retrieving the fault severities, prior work typically uses a simplified version of APFDc that only takes into consideration the test execution time [10, 15]. Our evaluation utilizes this simplified version of APFDc in our evaluation:

$$APFDc = \frac{\sum_{i=1}^m (\sum_{j=TF_i}^n t_j - \frac{1}{2} t_{TF_i})}{\sum_{j=1}^n t_j \times m}.$$

In the formula,  $n$ ,  $m$ , and  $TF_i$  have the same definitions as in the formula for APFD, while  $t_j$  is the execution time of the  $j$ th test.

Prior TCP work [10, 36] has shown that the classic TCP evaluation metric APFD can be very biased and unreal. More specifically, a technique that results in high APFD does not necessarily result in high APFDc, thus being not cost-efficient. Recently, Chen *et al.* [10] also showed that a simple cost-only technique outperforms coverage-based TCP techniques. In this work, we measure TCP effectiveness using both metrics, in particular evaluating IR-based TCP, which was only evaluated using APFD in the past [53]. We measure both as to compare how the two metrics can lead to different results in which TCP technique is the best.

**4.1.3 Failure-to-Fault Mapping.** Note that APFD and APFDc are defined with respect to the number of *faults* as opposed to failed tests. Prior work evaluated using seeded faults and mutants (treated as faults), with an exact mapping from test failures to faults in the code. However, in our evaluation, we only have information concerning which tests failed, and we do not have an exact mapping from failures to faults. Potentially, a single failed test can map to multiple faults in the code, and conversely multiple failed tests can all map to a single fault. In general, a developer would not know this exact mapping without deep investigation, and given our use of a dataset using real test failures, we also do not know the exact mapping. When answering our research questions, we assume that each single failed test maps to a single distinct fault, similar to one of the mappings Shi *et al.* used in their prior work on test-suite reduction [56], so the number of faults is then the number of test failures. In Section 5.5, we evaluate the impacts of a new mapping that all failures map to the same fault (only one fault in the code), the other extreme for mapping failures to faults.

### 4.2 Other TCP Techniques

We implement two simple TCP techniques, based on test execution time or historical failures, as baselines to compare IR-based TCP against. While these baselines are relatively simple, prior work has also shown them to be quite effective TCP techniques [10, 14]. We also implement four traditional coverage-based TCP techniques for comparison purposes as well. In general, all these other TCP techniques prioritize based on some metric. For ties in the metric, like with our implementation IR-based TCP, we order the tests deterministically in the order they would have been scheduled to run without any TCP technique in use.

**4.2.1 QTF.** A simple way to prioritize tests is based on the execution time of the tests. The intuition is that if the fastest-running tests can already find bugs, then running them first should lead to quicker detection of bugs. *QTF* prioritizes the tests by the ascending order of their execution time in the previous job, running the quickest test first. Chen *et al.* found this technique to be a strong baseline in their prior work [10].

**4.2.2 HIS.** Both industry [35, 40] and academia [7, 42, 59] have considered historical test failures when optimizing software testing techniques. In fact, Maven Surefire, the default unit testing plugin for Maven builds, has a built-in option for specifying that recently failed tests be run first [5]. The intuition is that more frequently failed tests are more likely to fail again in the future. *HIS* builds upon that intuition and prioritizes the tests based on the number of times the test has failed in prior jobs.

To implement *HIS*, we collect the historical failure frequency data for each test in each job by counting the number of times that the specific test has ever failed before the specific job. Each test  $t_i$  is then assigned a historical failure frequency  $h_{fi}$ , indicating the number of times it failed before, and *HIS* orders the tests by the descending order of  $h_{fi}$ .

**4.2.3 Coverage-Based TCP.** Coverage-based TCP techniques have been extensively evaluated in empirical studies [31, 50] and are widely used as the state-of-the-art techniques to compare new techniques against [44, 53]. We focus on four traditional coverage-based TCP techniques: (1) Total Test Prioritization [50], (2) Additional Test Prioritization [50], (3) Search-Based Test Prioritization [30, 31], and (4) Adaptive Random Test Prioritization (ARP) [21]. These four techniques are all change-unaware, cost-unaware TCP techniques that aim to find an ideal ordering of the tests based on the coverage information of code elements (statements/branches/methods) [31] obtained on the initial version. We utilize Lu *et al.*'s implementation of the four coverage-based TCP techniques [31].

For every build in our dataset, we collect coverage on the corresponding previous build. Prior work has found that the distance between the version for coverage collection and the version for test prioritization can greatly impact test prioritization results, as the tests and coverages would change over software evolution [13, 31]. In our evaluation, we compute the coverage at the build right before the build where the prioritized test ordering is evaluated, thus maximizing the effectiveness of the coverage-based techniques. We evaluate coverage-based TCP at its best for comparison purposes against IR-based TCP.

To collect coverage on each previous build, we download the corresponding previous build and try to rebuild it locally. We use a clean Azure virtual machine installed with Ubuntu 16.04, and we utilize the default Maven configuration to run the tests. We use OpenClover [43], a coverage collection tool, to collect the coverage information. OpenClover collects coverage per each test method, and we merge the coverage of each test method in the same test class together to form the coverage for each test class. Note that our local build might be different from the original build on Travis CI because of different Maven configurations, and the differences can make the local Maven build crash or produce no result, so we are unable to rerun all the jobs in our dataset locally. We also have to filter out the tests that are executed locally but not recorded in the original build

log, to make the test suite consistent between our local build and the original build run on Travis CI. (Such problems with building also illustrate difficulties in implementing and deploying an actual coverage-based TCP technique to run during regression testing.) We successfully rebuild 565 jobs across 49 projects, out of the 2,980 jobs in the dataset. When we compare these coverage-based TCP techniques against IR-based TCP, we compare them only on this subset of jobs.

### 4.3 Flaky Tests

Our evaluation of TCP using real test failures from historical test executions runs the risk of *flaky tests*. Flaky tests are tests that can non-deterministically pass or fail even for the same code under test [32]. The effect of flaky tests can be substantial, especially for change-aware TCP techniques. If a test failure is due to flakiness, the failure may have nothing to do with the recent change, and a change-aware TCP technique may (rightfully) rank such a test lower. However, if measuring TCP effectiveness based on test failures, a TCP technique can be evaluated as worse than what it should be.

To study the effect of flaky tests on TCP evaluation, we build a dataset that splits test failures into two groups: those definitely due to flaky tests and those likely due to real regressions introduced by program changes. Identifying whether a test failure is due to a flaky test is challenging and would in the limit require extensive manual effort [8, 26]. We use an automated approach by re-running the commits for the jobs with failed tests multiple times and checking what tests fail in those reruns. We select 252 jobs that contain exactly one single test failure and with a build time of less than five minutes, which we rerun six times on Travis CI using exactly the same configuration as when the job was originally run. (We choose only single test failures as to simplify our analysis of categorizing a failure as due to flakiness and a short build time as to not abuse utilizing Travis CI resources.) After the reruns, we record whether the single failure re-occurred or not. A test that does not consistently fail is definitely flaky, while one that does fail in all reruns is likely to be a failing test that indicates a real bug. (Note that having a consistent failure in six reruns does *not* necessarily mean the test is non-flaky.) For a job that we rerun, if the only failure in it is flaky, that job is a *flaky job*; otherwise, it is a *non-flaky job*. As a result, we collect 29 flaky jobs and 223 non-flaky jobs, and then we compare the effectiveness of TCP techniques on them.

## 5 EMPIRICAL EVALUATION

Our evaluation aims to answer the following research questions:

- **RQ1:** How do different information retrieval configurations impact IR-based TCP techniques in real software evolution?
- **RQ2:** How do IR-based TCP techniques compare against other TCP techniques, in terms of both cost-unaware and cost-aware metrics?
- **RQ3:** How can we further enhance IR-based TCP techniques?
- **RQ4:** How do flaky tests impact the evaluation of TCP in real software evolution?

## 5.1 RQ1: IR Configurations

The goal of this RQ is to explore different configurations for IR-based TCP; we then use the configuration that has the best performance as the default implementation of the IR-based TCP technique for the later RQs. When comparing the different configurations, we use APFD as the evaluation metric. IR-based TCP is fundamentally cost-unaware, so measuring APFD would better reflect the differences in quality due to differences in the configurations.

As described in Section 2, there are three main configuration options we need to set when implementing an IR-based TCP technique: (1) the code preprocessing approach for data objects and queries (by default using  $High_{token}$ ), (2) the amount of context to the program changes for constructing the query (by default using the whole file as the context), and (3) the retrieval model (by default using BM25, used in prior work [53]). We evaluate which values to set for each option in RQ1.1, 1.2 and 1.3, respectively.

**5.1.1 RQ1.1: Code Preprocessing.** We apply the four types of code preprocessing, *Low*,  $Low_{token}$ , *High*,  $High_{token}$ , to the tests and program changes for each failed job collected in our dataset. While we change the code preprocessing approach, we set the other parts to their default values. In other words, we use the whole-file strategy for the amount of context to the program changes and use BM25 as the retrieval model.

**Table 1: Comparing different preprocessing approaches**

Method	APFD			Time on IR (s)		Avg. v-length	
	mean	median	group	mean	median	data	query
<i>Low</i>	0.696	0.757	C	1.294	0.127	10419	3309
$Low_{token}$	0.755	0.849	A	2.194	0.326	1753	928
<i>High</i>	0.737	0.831	B	0.198	0.044	3056	677
$High_{token}$	0.752	0.844	AB	0.558	0.160	1011	358

Table 1 shows the effectiveness of IR-based TCP when changing the preprocessing approach measured in APFD (we show mean and median), the time spent on running the IR phase, and the mean vocabulary size, which is the number of distinct tokens in the data objects and in the query. Also, we perform a Tukey HSD test [61] for the APFD values, to see if there is any statistical difference between the approaches; the results are shown under the “group” column. The capital letters A-D are the results of the Tukey HSD test, where techniques are clustered into the letter groups, and A is the best while D is the worst (and having two letters means the technique is in a group that is between the two letter groups). We highlight the best approach in gray for each metric, so the highlighted boxes have either the highest mean or median APFD, the shortest mean or median time for running the IR phase, or the shortest vocabulary length for the data objects or queries.

From the table, we make the following observations. First, code tokenization is effective at reducing the vocabulary length. For example, comparing  $Low_{token}$  against *Low*, code tokenization reduces the vocabulary size of the data objects by 83.2%; comparing  $High_{token}$  against *High*, it reduces the vocabulary size of the data objects concerning extracted identifiers by 66.9%. Second, the approaches utilizing code tokenization usually take longer to run because the time complexity of the BM25 technique is linear to the number of words in the query. That is, although code tokenization

reduces the vocabulary size, it makes the data objects and queries longer by splitting long variable names into shorter tokens.<sup>3</sup> Finally,  $Low_{token}$ , *High*, and  $High_{token}$  lead to similar APFD values, while *Low* has clearly worse results than the other three approaches. The Tukey HSD test shows that  $Low_{token}$  performs the best, with it being in its own group and at the top (A). However,  $High_{token}$  performs almost as good (and ends up in the second-highest group, AB).  $High_{token}$  also runs faster and leads to smaller vocabulary lengths than  $Low_{token}$ . Overall, the  $High_{token}$  approach is slightly worse than  $Low_{token}$  for APFD, but it seems to provide the best trade-offs between APFD, time, and space among all four approaches.

**Table 2: Comparing different change contexts**

Context	APFD			# Ties	
	mean	median	group	mean	median
<i>0 line</i>	0.697	0.788	C	35.3	2
<i>1 line</i>	0.703	0.788	BC	30.9	2
<i>3 lines</i>	0.713	0.799	BC	25.8	1
<i>5 lines</i>	0.717	0.802	B	23.4	1
<i>Whole-file</i>	0.752	0.844	A	5.3	0

**5.1.2 RQ1.2: Context of Change.** Table 2 shows the evaluation result of how different lines of context influences the APFD and the number of ties. (Like with evaluating code preprocessing configuration, we set the other configurations to the default values, i.e., code preprocessing is set to  $High_{token}$  and the retrieval model is BM25.) Using the whole file as the context has the dominantly best performance in terms of both the highest APFD and the lowest rate of ties, whereas using 0 lines of context leads to the worst performance. We note that the mean value for number of ties is large relative to the median number of ties because of the few jobs that have an extremely large number of ties, which drags the mean up. In general, we observe that IR-based TCP performs better when there are more lines of context in the query.

One interesting observation is that when the length of the query is small, the IR scores for data objects tend to be identical (usually 0), so the prioritization barely changes the ordering of the tests from the original order. Similarity ties are common when we include 0 lines of context into the query. Ideally, the amount of context to include should lead to high APFD values but with a low rate of ties.

**5.1.3 RQ1.3: Retrieval Model.** We compare the four retrieval models described in Section 2.3. To choose an appropriate retrieval model, we perform a general evaluation of the overall mean and median APFD value across all jobs using each retrieval model. We also perform a per-project evaluation to see for each retrieval model the number of projects that perform the best (having the highest mean APFD across their jobs) with that model.

Table 3 shows the comparison between the different retrieval models. In this table and in the following tables, #B.Projs shows the number of projects that have the highest averaged metric value for each specific technique. We can see that BM25 model has both the highest mean and median APFD, and over 60% of the projects in our evaluation have the best results when using BM25.

<sup>3</sup>Note that we do not remove duplicated tokens, because prior work has found doing so would lead to worse results [53].

**Table 3: Comparing different retrieval models**

Retrieval Model	APFD			
	mean	median	group	#B.Projs
Tf-idf	0.728	0.812	B	27
BM25	0.752	0.844	A	75
LSI	0.707	0.775	C	13
LDA	0.659	0.735	D	8

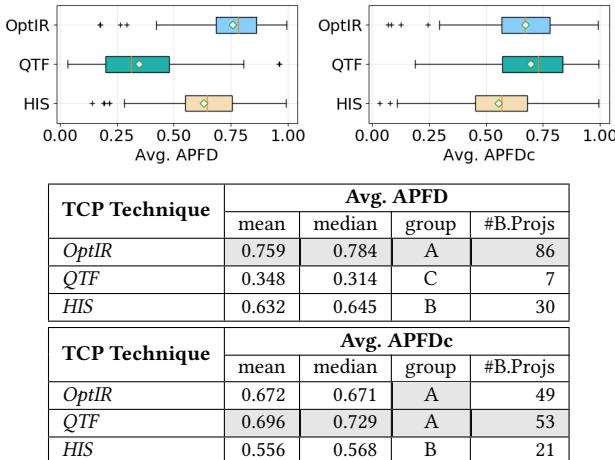
These results shows that BM25 performs better than Tf-idf, indicating that when two tests are equally similar to the same code change, running the test with the fewer lines of code early detects failures faster. Another important observation is that the vector space models (Tf-idf and BM25) perform better than the topic models (LSI and LDA) in terms of APFD, which is consistent with the conclusions found in previous work [52, 67]. The reason for this difference between types of models is likely because the topic models need many more tests to train on. Another likely reason is that the individual tests are not distinct enough between each other, making it hard for the models to generate effective topics.

**RQ1:** Overall, we determine that the best configuration options to set of IR-based TCP is to use *High<sub>token</sub>* as the code preprocessing approach, to use the whole file as the context to include in the queries, and to use BM25 as the retrieval model.

For the following RQs, we evaluate IR-based TCP using the recommended configurations. We denote IR-based TCP under these optimized configuration values as **OptIR**.

## 5.2 RQ2: Comparison of IR-Based TCP against Other TCP Techniques

For this RQ, we compare the effectiveness of OptIR against the other TCP techniques (Section 4.2). We measure both APFD and APFDC for evaluating the effectiveness of TCP.

**Figure 1: Comparing OptIR, QTF, and HIS APFD/APFDC**

**5.2.1 Comparison with QTF and HIS.** Figure 1 shows the results of the comparison between OptIR with QTF and HIS in terms of APFD and APFDC. The box plots show the distribution of average APFD and APFDC values per project for each TCP technique. The table in Figure 1 shows the mean and median for the average APFD and APFDC values across projects for each technique. The table also includes the number of projects where that technique results in the best average APFD or APFDC value. Furthermore, we perform a Tukey HSD test to check for statistically significant differences between the three techniques.

From the figure, we observe that the cost-unaware metric APFD can severely over-estimate the performance of cost-unaware techniques and under-estimate the performance of cost-aware techniques. If we take the mean APFDC value as the baseline, the mean APFD over-estimates the performance of OptIR by 12.9% and under-estimates the performance of QTF by 50.0%. Also, although QTF performs the worst when evaluated by APFD, it becomes better than both OptIR and HIS when evaluated by APFDC. The mean and median APFDC values for QTF are slightly higher than for OptIR, although the Tukey HSD test suggests that they belong in the same group (A). We further perform a Wilcoxon pairwise signed-rank test [63] to compare the pairs of APFDC values per project between OptIR and QTF, and we find the *p*-value is 0.339, which does not suggest statistically significant differences between the two. We cannot say then that OptIR is superior to QTF (especially since QTF has higher mean and median APFDC values). Concerning HIS, we clearly see it does not perform well in terms of APFDC. The main reason is because 45.1% of the failures in our dataset are first-time failures for a test, so there is often not any history for HIS to use.

**5.2.2 Comparison with Coverage-Based TCP Techniques.** In Figure 2, we further compare against traditional coverage-based TCP techniques (Section 4.2.3). The figure shows box plots and a table similar to in Figure 1, except it also includes coverage-based TCP techniques. Recall that we could only collect the test coverage information on a subset of the full dataset (565 jobs across 49 projects), so the results for OptIR, QTF, and HIS shown are different from what are shown in the previous figure. Besides the mean and median averaged APFD/APFDC by each project, we further perform a Tukey HSD test to check for statistically significant differences between all seven TCP techniques. Note that the sum of all the values for #B.Projs does not necessarily add up to the total 49 projects for this table. The reason is that when there is a tie between techniques for the best APFD/APFDC value for a project, we count that project for all the techniques that tie.

From the results, we observe that OptIR, even in this smaller dataset, still performs better than QTF and HIS based on APFD. OptIR also performs better than coverage-based techniques by APFD as well, based on the higher mean and median values. However, while the APFD value is higher, the Tukey HSD test shows that OptIR is in the same (A) group as all the coverage-based TCP techniques, except for ARP (which performs worse). Furthermore, like before, QTF performs the worst based on APFD.

However, when we consider APFDC, once again, QTF outperforms the other TCP techniques, including the coverage-based ones. QTF has the highest mean and median APFDC values, and the Tukey HSD test shows QTF in its own group (A) that is above the groups

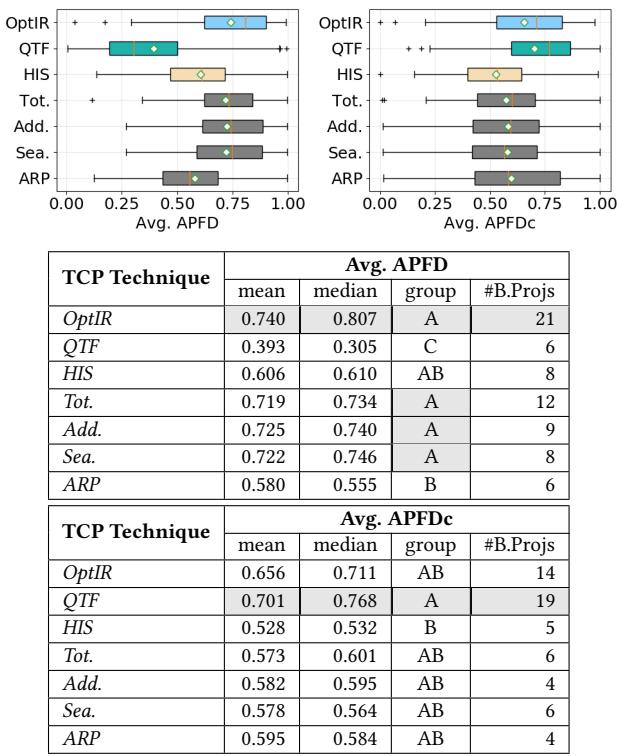


Figure 2: Comparing all TCP techniques’ APFD/APFDc

designated for all the other techniques. When we do a series of Wilcoxon pairwise signed-rank tests between QTF APFDc values and the values for all the other techniques, we see that the differences are statistically significant for all except for OptIR, once again. Also, OptIR ends up in the same Tukey HSD group as all the coverage-based TCP techniques.

Another observation we make is that OptIR in general suffers from a smaller performance loss when evaluation is changed from using APFD to APFDc, compared to coverage-based TCP techniques. The reason is that high-coverage tests tend to be more time-consuming, while higher IR-similarity tests related to the changes do not have such a tendency. This tendency makes traditional coverage-based TCP techniques have relatively lower APFDc values, as more costly tests are prioritized earlier. The ARP technique prioritizes tests based on the diversity of the test coverages rather than the amount of test coverages, so the difference from switching to APFDc is not as prevalent for this technique as well.

**RQ2:** QTF outperforms almost all other techniques, including traditional coverage-based TCP techniques, suggesting that future work on improving TCP should be compared against and at least perform better than QTF. OptIR performs similarly as QTF when measured by APFDc. OptIR also does not suffer much of a difference in performance when switching the evaluation metric from APFD to APFDc.

### 5.3 RQ3: Hybrid Techniques

From RQ2, we find that OptIR does not outperform the simple QTF technique based on APFDc, so we aim to improve OptIR by making it cost-aware. Prior work has tried to address the cost-inefficiency of coverage-based TCP techniques by implementing Cost-Cognizant coverage-based techniques that balance the coverage and the test-execution time, e.g., state-of-the-art Cost-Cognizant Additional Test Prioritization prioritizes tests based on the additional coverage per unit time [10, 15, 36]. However, recent work found that these techniques perform similarly or even worse than the cost-only technique, i.e., QTF. The possible reason is that traditional coverage-based TCP techniques tend to put more costly tests (those with more coverage) earlier, which contrasts with the goals of the Cost-Cognizant feature that tends to put less costly tests earlier. The change-aware IR techniques we study are based on the textual similarity and do not give stronger preference to complicated and costly tests. Therefore, we expect that IR techniques potentially perform better when hybridized with QTF. We also consider hybridizing with HIS to consider historical failure frequencies as well.

Before we hybridize OptIR to consider test execution cost or historical failure frequencies, we first consider a hybrid technique combining just test execution cost and historical failures. For a test  $T_i$ , given its historical failure frequency  $h_{f_i}$  and its prior execution time  $t'_i$ , we define a TCP technique that rearranges tests by the descending order of:

$$a_{cch}(T_i) = h_{f_i}/t'_i \quad (1)$$

We denote this technique as **CCH**. We use CCH as another comparison against the hybrid techniques that directly improve OptIR.

We design the first Cost-Cognizant IR-based TCP technique that orders tests based on their execution cost and IR score. For a test  $T_i$ , given its IR score  $ir_i$  (computed using the same optimized configuration values as with OptIR) and its execution time  $t'_i$  in the corresponding previous job, we define a TCP technique that rearranges tests by the descending order of:

$$accir(T_i) = ir_i/t'_i \quad (2)$$

We denote this technique as **CCIR**.

We next consider a hybrid technique that combines OptIR with HIS. For a test  $T_i$ , given its IR score  $ir_i$  and its historical failure frequency  $h_{f_i}$ , we define a TCP technique that rearranges tests by the descending order of:

$$a_{hir}(T_i) = (h_{f_i} * ir_i) \quad (3)$$

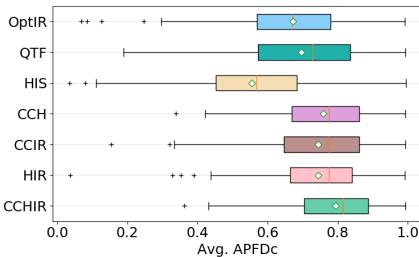
We denote this technique as **HIR**.

Finally, we consider a hybrid technique that combines both test execution cost and historical failures with OptIR. For a test  $T_i$ , given its IR score  $ir_i$ , its historical failure frequency  $h_{f_i}$ , and its prior execution time  $t'_i$ , we define a TCP technique that rearranges tests by the descending order of:

$$a_{cchir}(T_i) = (h_{f_i} * ir_i)/t'_i \quad (4)$$

We denote this technique as **CCHIR**.

Figure 3 shows our evaluation of the hybrid TCP techniques on the full dataset. From the figure, we see that all hybrid techniques are better than the individual non-hybrid techniques that they are built upon. For example, CCIR results in a better mean APFDc



TCP Technique	Avg. APFDc			
	mean	median	group	#B.Projs
OptIR	0.672	0.671	C	19
QTF	0.696	0.729	BC	9
HIS	0.556	0.568	D	5
CCH	0.758	0.776	AB	18
CCIR	0.744	0.776	AB	10
HIR	0.744	0.774	AB	31
CCHIR	0.794	0.815	A	47

Figure 3: Comparing against the hybrid TCP techniques

value than both OptIR (by 10.7%) and QTF (by 6.9%) from which this hybrid technique is constructed. Overall, we observe that the three-factor hybrid technique combining IR scores, prior execution time, and historical failure frequency outperforms any other technique we evaluate by at least 4.7% in terms of the mean averaged APFDc, which means the three factors we consider in the hybrid technique can enhance each other and form an effective hybrid technique.

To see whether there is a statistically significant difference between the distributions of the APFDc values of the different techniques, we once again perform a Tukey HSD test on the APFDc results for the three single techniques (OptIR, QTF, and HIS) and three hybrid techniques (CCIR, CCH, and CCHIR). We see a statistically significant difference between hybrid techniques and non-hybrid techniques, with CCHIR being the best and HIS being the worst. The results of these statistical tests suggest that the hybrid techniques, especially the three-factor hybrid technique CCHIR, are effective techniques to perform TCP.

Figure 4 shows the results of our per-project analysis comparing the different hybrid techniques. The figure shows the average APFDc of all jobs per project for each technique; the projects are sorted left to right by descending order of each one's average APFDc value for CCHIR. We do not include the per-project results of the non-hybrid techniques into the figure, because our prior overall evaluation on the full dataset shows that the non-hybrid techniques generally perform worse than the hybrid ones. The accompanying table in the figure shows the number of jobs and the average APFDc values across all those jobs per project for each hybrid technique. Once again, the best APFDc value is highlighted for each project.

Based on Figure 4, for most projects, CCHIR performs the best among all techniques or performs closely to the best technique, and it never performs the worst. Specifically, we perform a Tukey HSD test on the hybrid techniques on all 119 projects in our dataset that each has more than one job. We find that there are 105 projects for which CCHIR performs the best among all hybrid techniques, being in group A. Also, the standard deviation of the average APFDc

per project for CCHIR is the lowest among all studied techniques, demonstrating that CCHIR tends to be more stable. These results further suggest that developers should in general use CCHIR.

We further inspect the projects that CCHIR does not perform as well. Among all the 27 projects that the average APFDc value is lower than 0.7 with CCHIR, there are 18 of them where HIS performs the best. We inspect these projects and find that these projects are those where QTF also performs poorly. Therefore, if CCHIR performs poorly on a project, it is likely due to the test cost factor. On closer inspection of these projects, we find that the failing tests are those that run quite long, so both QTF and the hybrid techniques that hybridize with QTF tend to prioritize those tests later. This finding further demonstrates the necessity of designing more advanced techniques to better balance the textual, cost and historical information for more powerful test prioritization.

**RQ3:** We find that hybrid techniques that combine OptIR with test execution cost and historical failures improves over OptIR with respect to APFDc, with CCHIR, which utilizes all these factors, performing the best among all hybrid techniques.

#### 5.4 RQ4: Impacts of Flaky Tests

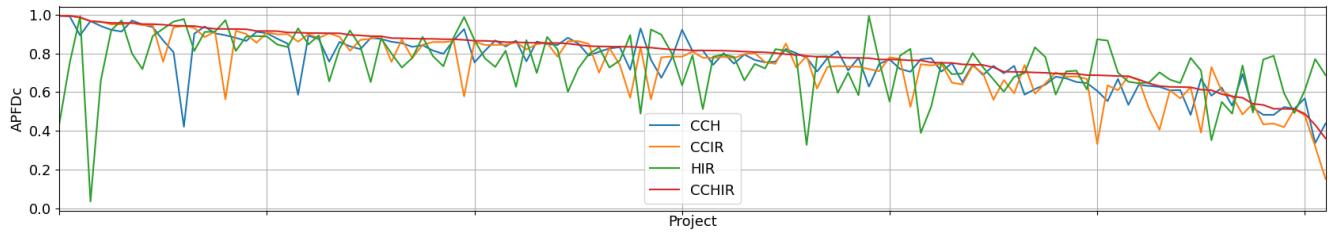
Figure 5 shows boxplots side-by-side comparing the APFDc for flaky jobs and non-flaky jobs under seven major TCP techniques (four change-aware IR techniques: Tf-idf, BM25, LSI, LDA; two change-unaware techniques QTF, HIS; and one hybrid technique CCHIR). We observe that all change-aware techniques (Tf-idf, BM25, LSI, LDA) perform better on non-flaky jobs than on flaky jobs. As flaky test failures are usually introduced into the program before the latest version [32] while non-flaky failures are more likely to be regressions introduced by the latest change, change-aware techniques are better at prioritizing non-flaky tests whose failures then likely indicate regressions in the program change.

We also find that QTF and HIS have better performance on flaky jobs than on non-flaky jobs. This result suggests that in our dataset flaky tests tend to run faster and are more likely to have failed in the past. Furthermore, CCHIR performs the best among the seven techniques on both flaky jobs and non-flaky jobs. This result demonstrates the robustness and effectiveness of our hybrid technique as it is positively influenced by the change-aware technique without suffering much from its weaknesses.

**RQ4:** Flaky test failures can affect the results from evaluating and comparing different TCP techniques. In particular, for change-aware techniques, these techniques would seem to perform better when evaluated using only non-flaky failures. However, we find that CCHIR still performs the best among all evaluated techniques regardless of flaky or non-flaky failures.

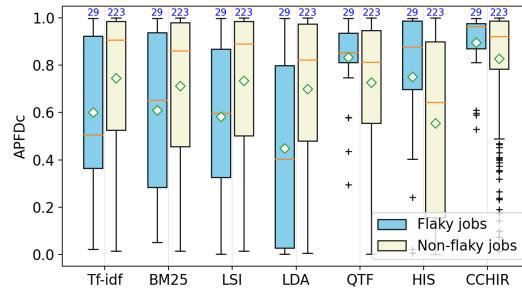
#### 5.5 Discussion

So far in our evaluation, we assume that each failure maps to a distinct fault. However, in reality it is possible that multiple test failures are due to the same fault in the source code, and with different failure-to-fault mappings the computed APFD and APFDc



Proj. Name	#Jobs	Avg. APFDc				Proj. Name	#Jobs	Avg. APFDc			
		CCH	CCIR	HIR	CCHIR			CCH	CCIR	HIR	CCHIR
zeroturnaround/zt-zip	3	0.994	0.994	0.439	0.994	flaxsearch/luwak	3	0.990	0.993	0.751	0.993
RIPE-NCC/whois	5	0.893	0.985	0.991	0.985	yandex-qatools/postgresql-embedded	2	0.966	0.966	0.037	0.966
mp911de/logstash-gelf	6	0.941	0.963	0.662	0.961	undera/jmeter-plugins	10	0.921	0.949	0.919	0.957
mitreid-connect/OpenID-Connect-Java-Spring-Server	5	0.912	0.950	0.969	0.957	apache/commons-compress	4	0.969	0.956	0.799	0.956
internetcharive/heritrix3	15	0.948	0.945	0.719	0.951	apache/incubator-dubbo	82	0.935	0.944	0.889	0.950
perwende/spark	9	0.867	0.757	0.927	0.948	wmixvideo/nfe	20	0.806	0.934	0.963	0.943
mjiderhamn/classloader-leak-prevention	1	0.422	0.941	0.976	0.941	apache/systemml	6	0.901	0.931	0.812	0.941
eclipse-vertx/vertx.x	95	0.938	0.884	0.910	0.933	vert-x3/vertx-web	8	0.903	0.912	0.912	0.926
ocpsoft/rewrite	42	0.893	0.564	0.971	0.926	graphhopper/jspirt	20	0.880	0.916	0.812	0.925
ebean-orm/ebean	44	0.864	0.900	0.886	0.924	xerthio/jedis	67	0.911	0.854	0.889	0.915
DiUS/java-faker	16	0.904	0.908	0.887	0.915	alibaba/fastjson	46	0.875	0.898	0.845	0.914
FasterXML/jackson-databind	44	0.850	0.899	0.833	0.908	HubSpot/jinjava	7	0.587	0.860	0.929	0.907
killme2008/aviator	4	0.891	0.899	0.846	0.904	resteasy/Resteasy	101	0.873	0.879	0.893	0.904
rhwayfun/spring-boot-learning-examples	4	0.757	0.904	0.656	0.904	protegeproject/protege	3	0.859	0.885	0.824	0.903
alipay/sofa-rpc	72	0.836	0.813	0.918	0.895	gresrun/jesqu	9	0.821	0.869	0.836	0.892
amzn/ion-java	11	0.879	0.875	0.652	0.888	dooApp/FXForm2	10	0.874	0.888	0.877	0.888
Angel-ML/angel	6	0.860	0.756	0.799	0.878	google/error-prone	26	0.853	0.871	0.727	0.878
prometheus/client_java	21	0.834	0.776	0.776	0.876	socketio/socket.io-client-java	4	0.842	0.844	0.885	0.875
orbit/orbit	5	0.814	0.858	0.786	0.875	openmrs/openmrs-core	48	0.798	0.858	0.733	0.873
ff4j/ff4j	9	0.868	0.863	0.883	0.872	yegor256/rultor	13	0.925	0.578	0.987	0.870
joelittlejohn/jonschema2pojo	9	0.753	0.861	0.868	0.865	basho/riak-java-client	39	0.813	0.843	0.774	0.864
JSQlParser/JSqlParser	34	0.866	0.844	0.730	0.862	elasticjob/elastic-job-lite	86	0.836	0.844	0.813	0.858
LiveRamp/hank	7	0.866	0.857	0.628	0.857	bootique/bootique	5	0.759	0.820	0.868	0.855
RoaringBitmap/RoaringBitmap	44	0.861	0.847	0.699	0.855	turdt/jdeb	4	0.847	0.853	0.884	0.853
junkdog/artemis-odb	12	0.843	0.783	0.820	0.853	lukas-krecan/JsonUnit	57	0.880	0.863	0.602	0.850
zendesk/maxwell	27	0.850	0.863	0.722	0.844	FasterXML/jackson-dataformat-xml	7	0.788	0.849	0.795	0.838
apache/incubator-druid	93	0.808	0.701	0.831	0.836	pf4j/pf4j	13	0.825	0.827	0.727	0.835
hs-web/hsweb-framework	25	0.832	0.740	0.763	0.834	apache/servicecomb-pack	35	0.716	0.571	0.894	0.833
SpigothMC/BungeeCord	1	0.928	0.831	0.489	0.831	davidmoten/rxjava-extras	12	0.767	0.563	0.923	0.829
stanford-futuredata/macrobase	16	0.674	0.779	0.896	0.824	debezium/debezium	61	0.776	0.784	0.801	0.820
alibaba/druid	5	0.921	0.783	0.635	0.818	eclipse/paho.mqtt.java	20	0.813	0.809	0.789	0.815
jtableaw/tableaw	34	0.815	0.777	0.513	0.815	google/closure-compiler	38	0.741	0.777	0.783	0.814
scabal/seyren	6	0.801	0.782	0.793	0.814	twitter/GraphJet	13	0.747	0.775	0.782	0.811
weibocom/motan	24	0.794	0.796	0.661	0.809	abel533/Mapper	12	0.766	0.802	0.746	0.807
rest-assured/rest-assured	7	0.753	0.754	0.722	0.805	keycloak/keycloak	52	0.757	0.746	0.822	0.802
apache/incubator-dubbo-spring-boot-project	20	0.822	0.851	0.813	0.795	st-st/st-sts	11	0.801	0.727	0.786	0.793
cglib/cglib	1	0.778	0.787	0.328	0.787	gchq/Gaffer	41	0.706	0.618	0.783	0.783
rapidoid/rapidoid	15	0.775	0.730	0.785	0.781	spring-projects/spring-data-redis	41	0.811	0.735	0.597	0.780
aws/aws-sdk-java	56	0.712	0.732	0.700	0.777	RipMeApp/ripme	74	0.777	0.731	0.584	0.777
qos-ch/logback	1	0.629	0.719	0.992	0.775	floodlight/floodlight	20	0.746	0.708	0.757	0.768
zhang-rf/mybatis-boost	19	0.768	0.779	0.551	0.768	etripcorp/apollo	31	0.720	0.771	0.788	0.768
sakaiproject/sakai	39	0.705	0.524	0.823	0.764	awslabs/amazon-kinesis-client	28	0.769	0.744	0.390	0.764
fakereplace/fakereplace	2	0.775	0.738	0.525	0.753	alibaba/jetcache	4	0.706	0.746	0.754	0.751
spring-projects/spring-data-mongodb	41	0.750	0.649	0.692	0.750	networkt/light-4j	25	0.651	0.640	0.697	0.743
ModeShape/modeshape	7	0.742	0.736	0.801	0.742	apache/incubator-pinot	52	0.689	0.696	0.728	0.740
searchbox-io/Jest	13	0.736	0.561	0.666	0.728	JanusGraph/janusgraph	144	0.697	0.662	0.602	0.705
AxonFramework/AxonFramework	48	0.736	0.594	0.679	0.705	jhy/jsoup	25	0.587	0.740	0.699	0.704
google/auto	13	0.616	0.593	0.831	0.701	spring-projects/spring-security-oauth	27	0.639	0.645	0.781	0.700
vipshop/viptools	21	0.680	0.701	0.588	0.696	teamed/quice	12	0.671	0.680	0.707	0.695
getheimdall/heimdall	15	0.652	0.681	0.711	0.695	winder/Universal-G-Code-Sender	11	0.647	0.668	0.615	0.688
jcab/iocabi-github	4	0.607	0.333	0.872	0.687	rickfast/consul-client	9	0.554	0.636	0.866	0.685
apache/rocketmq	49	0.668	0.610	0.700	0.683	demoselle/framework	6	0.535	0.682	0.654	0.682
pippo-java/pippo	11	0.638	0.662	0.646	0.665	vipshop/Saturn	42	0.632	0.514	0.650	0.645
apache/zeppelin	37	0.627	0.407	0.702	0.631	onelogin/java-saml	4	0.609	0.609	0.665	0.627
codelibs/fess	45	0.611	0.568	0.647	0.627	square/moshi	5	0.483	0.625	0.776	0.625
alipay/sofa-bolt	20	0.669	0.392	0.711	0.613	alexxiyang/shiro-redis	8	0.582	0.729	0.353	0.610
doanduyhai/Achilles	3	0.623	0.606	0.550	0.590	pholser/junit-quickcheck	3	0.530	0.579	0.489	0.579
redpen-cc/redpen	16	0.694	0.486	0.737	0.571	jaeksoft/opensearchserver	6	0.519	0.540	0.495	0.540
apache/servicecomb-java-chassis	49	0.484	0.434	0.768	0.534	imagefree/image	43	0.483	0.438	0.787	0.514
spring-projects/spring-data-cassandra	40	0.523	0.420	0.595	0.514	shrinkwrap/resolver	3	0.515	0.513	0.493	0.513
opensagres/xdocreport	6	0.568	0.481	0.610	0.489	jenkinsci/java-client-api	3	0.338	0.320	0.770	0.431
sismics/reader	7	0.439	0.154	0.688	0.362						

Figure 4: Comparing the hybrid TCP techniques by project



**Figure 5: Comparing different techniques on flaky/non-flaky jobs**

values could change. We evaluate again the major research questions by mapping all failures to the same fault to check whether the conclusions still remain the same. For RQ1, we calculate the mean and median APFD by job for Tf-idf, BM25, LSI and LDA; for RQ2, we calculate the the mean and median APFD and APFDc by project for OptIR, QTF, HIS, and the coverage-based techniques (only on the jobs in common among all the techniques); and for RQ3, we calculate the mean and median APFDc by project for OptIR, QTF, CCIR, HIS, CCH, CCHIR. For all these values, we recalculate while considering all failures mapping to the same fault. In other words, all we want to observe is how well the different TCP techniques order tests such to get a failed test to come as early as possible. We do not re-evaluate RQ4, concerning flaky tests, because our evaluation for RQ4 already only considers jobs with one test failure.

We compare the results with mapping each failure to a distinct fault (the type of evaluation we use before for all prior RQs) to see if there is a difference in overall results. Table 4 shows the evaluation results. From the table, we observe that although the APFD and APFDc values are larger when mapping all failures to one fault than when mapping each failure to distinct faults, different failure-to-fault mappings lead to similar overall conclusions.

The only result inconsistent with what we observe from before is that, after mapping all failures to the same fault, Additional Test Prioritization and Search-Based Test Prioritization outperform OptIR when evaluated by mean APFD. The potential reason is that both Additional Test Prioritization and Search-Based Test Prioritization care only about the additional coverage. Therefore, when there are multiple failures mapping to the same fault, the corresponding fault-revealing methods will grant only one of the failures a high priority. That is, the APFD and APFDc values are more under-estimated in these techniques than in other techniques. However, in terms of APFDc, the cost-aware metric, OptIR still outperforms the coverage-based techniques. Overall, all other conclusions concerning trends remain the same, i.e., BM25 is the best retrieval model for IR-based TCP, QTF outperforms all other non-hybrid techniques based on APFDc, and the hybrid CCHIR technique performs the best among all the hybrid and non-hybrid techniques.

## 6 THREATS TO VALIDITY

For our dataset, we extract test FQNs, test outcomes, and test execution times from job logs, so whether we can get a complete

**Table 4: Impact of different failure-to-fault mappings**

RQ No.		One to One		All to One	
		APFD	mean	median	mean
RQ1	Tf-idf	0.728	0.812	0.771	0.885
	BM25	0.752	0.844	0.795	0.913
	LSI	0.707	0.775	0.753	0.860
	LDA	0.659	0.735	0.713	0.837
	Avg. APFD	0.740	0.807	0.768	0.838
RQ2	QTF	0.393	0.305	0.440	0.397
	HIS	0.606	0.610	0.650	0.647
	Tot.	0.719	0.734	0.756	0.785
	Add.	0.725	0.740	0.774	0.802
	Sea.	0.722	0.746	0.771	0.796
	ARP	0.580	0.555	0.633	0.603
	Avg. APFDc	0.656	0.711	0.692	0.770
	OptIR	0.701	0.768	0.733	0.806
RQ3	HIS	0.528	0.532	0.578	0.584
	Tot.	0.573	0.601	0.622	0.678
	Add.	0.582	0.595	0.637	0.672
	Sea.	0.578	0.564	0.634	0.665
	ARP	0.595	0.584	0.649	0.673
	Avg. APFDc	0.672	0.671	0.727	0.730
	OptIR	0.696	0.729	0.733	0.778
	QTF	0.556	0.568	0.606	0.635
RQ4	HIS	0.758	0.776	0.793	0.826
	CCH	0.744	0.776	0.779	0.804
	CCIR	0.744	0.774	0.790	0.816
	HIS	0.794	0.815	0.826	0.853
	CCHIR	0.794	0.815	0.826	0.853

and accurate dataset is dependent on the completeness and the parsability of log files.

Our implementations of the TCP techniques and our experimental code may have bugs that affect our results. To mitigate this threat, we utilize well-developed and well-used tools, such as Travis CI to parse job logs, OpenClover to collect code coverage, javalang [60] to parse the AST of source code, and various mature libraries to compute IR similarity scores. Furthermore, we tested our implementations and scripts on small examples to increase confidence in the correctness of our results.

For our hybrid techniques, we collect the historical failures from the failed jobs in our dataset, which satisfy our requirements as discussed in Section 3, as opposed to failures from all possible failed jobs on Travis CI. As such, the historical failures we use is a subset of all possible failures for tests as recorded in Travis CI, which is in turn a subset of all possible failures that occurred during development (failures occurred during local development and not pushed for continuous integration). However, developers may not need to see *all* failures, as a recent study on evaluating test selection algorithms at Google focused on only the transition of test statuses (e.g., going from pass to fail), instead of just straight failures (developers may not have addressed the fault yet) [29]. We also construct our dataset using only jobs where the currently failing tests were not failing in the job before (Section 3).

The metrics for evaluating TCP techniques are also crucial for this work. We adopt both the widely used cost-unaware metric APFD and the cost-aware metric APFDc for evaluating all the studied TCP techniques.

## 7 RELATED WORK

Regression testing [64] has been extensively studied in the literature [13, 16, 17, 28, 49, 56, 57, 65]. Among various regression testing techniques, the basic idea of test-case prioritization (TCP) techniques is to prioritize tests that have a higher likelihood of detecting bugs to run first. Most prior work has implemented techniques based on test coverage (prioritizing tests that cover more) and diversity (ordering tests such that similar tests in terms of coverage are ordered later), and investigated characteristics that can affect TCP effectiveness such as test granularity or number of versions from original point of prioritization [19, 21, 30, 31, 33, 66]. Recent work shows that traditional coverage-based techniques are not cost effective at running the tests that detect bugs earlier, because they tend to execute long-running tests first [10]. To address this problem, one solution is to make new Cost-Cognizant coverage-based techniques that are aware of both the coverage and cost of tests [10, 36]. However, Chen *et al.* [10] showed that the Cost-Cognizant Additional Test Prioritization technique performed better than the cost-only technique for only 54.0% of their studied projects, and the mean APFDc of the Cost-Cognizant Additional Test Prioritization technique is just 3.0% better than that of the cost-only technique. Another solution is to utilize machine learning techniques, but it requires non-trivial training process involving a large number of historical bugs that are often unavailable [10]. In our work, we propose new simplistic hybrid TCP techniques that combine an IR-based TCP technique with test execution cost and historical failure frequency. We find that these hybrid techniques perform better, leading to higher APFDc values.

Prior work has utilized information retrieval (IR) techniques in software testing [24, 41, 53]. Our work is most similar to prior work by Saha *et al.*, who prioritized tests in Java projects using IR techniques, ordering tests based on the similarities between tests and the program changes between two software versions [53]. Using IR to perform TCP can be effective, because it does not require costly static or dynamic analysis, only needing lightweight textual-based computations. Saha *et al.* found that their IR-based TCP technique outperforms the traditional coverage-based techniques. However, Saha *et al.* evaluated their technique on a dataset with only 24 manually-constructed version pairs from eight projects to reproduce regressions, which potentially makes their result noninclusive and less generalizable. Also, they evaluate using APFD, which does not consider the cost of testing and does not reflect the effectiveness of TCP as well as using APFDc. Our work improves upon Saha *et al.*'s evaluation by introducing a much larger dataset with real failures and real time, and we utilize APFDc to evaluate the effectiveness of IR-based TCP techniques. Mattis and Hirschfeld also recently studied IR-based TCP on Python code and tests [38]. They propose a new IR-based technique based on the predictive power of lexical features in predicting future failures. Mattis and Hirschfeld find that their predictive model outperforms BM25 based on APFD

measured through seeded mutants and faults in the dynamically-typed programming domain of Python, but their new technique does not outperform BM25 when time to first failure is considered.

Recently, some other work has used Travis CI to construct large and real datasets to evaluate testing techniques [8, 20, 25, 39, 56]. We also collect our dataset from Travis CI, collecting information from builds that involve real changes from developers and real test failures. With real failures used in evaluation, our work improves upon evaluation of TCP in prior work, which utilized mutation testing [7, 10, 23]. Mattis *et al.* in a parallel, independent work published a dataset for evaluating TCP [6, 39]. For their dataset, Mattis *et al.* also collected test failures from Travis CI, collecting more than 100,000 build jobs across 20 open-source Java projects from GitHub. Their dataset contains the test outcomes and test times per job, but it does not contain the source code diffs we need to perform IR-based TCP.

There has been much prior work in the area of flaky tests [8, 18, 26, 27, 32, 55, 58]. In our work, we evaluate how TCP techniques perform on flaky and non-flaky tests separately. From our evaluation, we find that the change-aware IR-based TCP techniques perform better when considering only non-flaky test failures. However, we find that the hybrid TCP technique CCHIR is quite robust in the face of flaky tests.

## 8 CONCLUSIONS

In this work, we conduct an empirical evaluation of TCP techniques on a large-scale dataset with real software evolution and real failures. We focus on evaluating IR-based TCP techniques, and our results show that IR-based TCP is significantly better than traditional coverage-based techniques, and its effectiveness is close to that of the cost-only technique when evaluated by APFDc, a cost-aware metric. We also implement new hybrid techniques that combine change-aware IR with historical test failure frequencies and test execution time, and our hybrid techniques significantly outperform all the non-hybrid techniques evaluated in this study. Lastly, we show that flaky tests have a substantial impact on the change-aware TCP techniques.

## ACKNOWLEDGMENTS

We thank Wing Lam, Owolabi Legunsen, and Darko Marinov for their extensive discussion with us on this work. We acknowledge support for research on test quality from Futurewei. We also acknowledge support from Alibaba. This work was partially supported by NSF grants CNS-1646305, CCF-1763906, and OAC-1839010.

## REFERENCES

- [1] [n.d.] GitHub. <https://github.com/>.
- [2] [n.d.] Maven. <http://maven.apache.org/>.
- [3] [n.d.] Travis-CI. <https://travis-ci.org/>.
- [4] 2020. Empirically Revisiting and Enhancing IR-based Test-Case Prioritization. <https://sites.google.com/view/ir-based-tcp>.
- [5] 2020. Maven Surefire Plugin - surefire:test. <https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html>.
- [6] 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. <https://zenodo.org/record/3712290>.
- [7] Maral Azizi and Hyunsook Do. 2018. ReTEST: A cost effective test case selection technique for modern software development. In *ISSRE*.
- [8] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tiffany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*.

- [9] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *MSR*.
- [10] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *ESEC/FSE*.
- [11] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *JASIS* 41, 6 (1990).
- [12] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*.
- [13] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *TSE* 28, 2 (2002).
- [14] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *FSE*.
- [15] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritization. In *ISSTA*.
- [16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *ICSE Demo*.
- [17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*.
- [18] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*.
- [19] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *ICSE*.
- [20] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *ASE*.
- [21] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *ASE*.
- [22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA Demo*.
- [23] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*.
- [24] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. 2014. Test case prioritization based on information retrieval concepts. In *APSEC*, Vol. 1.
- [25] Adriana Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*.
- [26] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhi, and Suresh Thumalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*.
- [27] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakes: A framework for detecting and partially classifying flaky tests. In *ICST*.
- [28] Owolabi Legunse, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *FSE*.
- [29] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing transition-based test selection algorithms at Google. In *ICSE-SEIP*.
- [30] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search algorithms for regression test case prioritization. *TSE* 33, 4 (2007).
- [31] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *ICSE*.
- [32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
- [33] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *FSE*.
- [34] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. 2018. Assessing test case prioritization on real faults and mutants. In *ICSCME*.
- [35] Mateusz Machalica, Alex Samylinkin, Meredith Porth, and Satish Chandra. 2018. Predictive test selection. In *ICSE-SEIP*.
- [36] Alexey G. Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-cognizant test case prioritization*. Technical Report. Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln.
- [37] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2010. Introduction to information retrieval. *Nat. Lang. Eng.* 16, 1 (2010).
- [38] Toni Mattis and Robert Hirschfeld. 2020. Lightweight Lexical Test Prioritization for Immediate Feedback. *Programming Journal* 4 (2020).
- [39] Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An open-source dataset for evaluating regression test prioritization. In *MSR*.
- [40] Atif Memori, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE-SEIP*.
- [41] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 2011. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *ICWS*.
- [42] Tanzeem Bin Noon and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE*.
- [43] Marek Parfianowicz and Grzegorz Lewandowski. 2017–2018. OpenClover. <https://openclover.org>.
- [44] David Paterson, José Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An empirical study on the use of defect prediction for test case prioritization. In *ICST*.
- [45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Mathieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine learning in Python. *JMLR* 12 (2011).
- [46] Radim Rehůřek and Petr Sojka. 2010. Software framework for topic modelling with large corpora. In *LREC*.
- [47] Stephen Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Ret* 3, 4 (2009).
- [48] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. 2000. Experimentation as a way of life: Okapi at TREC. *Inf. Process. Manage.* 36, 1 (2000).
- [49] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *TSE* 22, 8 (1996).
- [50] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *ICSM*.
- [51] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *TSE* 27, 10 (2001).
- [52] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *ASE*.
- [53] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *ICSE*.
- [54] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.* 24, 5 (1988).
- [55] August Shi, Alex Gyori, Owolabi Legunse, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*.
- [56] August Shi, Alex Gyori, Suleiman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*.
- [57] August Shi, Milica Hadži-Tanović, Lingming Zhang, Darko Marinov, and Owolabi Legunse. 2019. Reflection-aware static regression test selection. *PACMPL* 3, OOPSLA (2019).
- [58] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakes: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*.
- [59] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA*.
- [60] Chris Thunes. 2018. c2nes/javalang. <https://github.com/c2nes/javalang>.
- [61] John W. Tukey. 1949. Comparing individual means in the analysis of variance. *Biometrics* 5, 2 (1949).
- [62] Xing Wei and W. Bruce Croft. 2006. LDA-based document models for ad-hoc retrieval. In *SIGIR*.
- [63] Frank Wilcoxon. 1945. Individual comparisons by ranking methods.
- [64] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *STVR* 22, 2 (2012).
- [65] Lingming Zhang. 2018. Hybrid regression test selection. In *ICSE*.
- [66] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*.
- [67] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *ICSE*.

# Intermittently Failing Tests in the Embedded Systems Domain

Per Erik Strandberg  
Westermo Network Technologies AB  
Sweden  
Mälardalen University  
Sweden

Wasif Afzal  
Mälardalen University  
Sweden

Thomas J. Ostrand  
Mälardalen University  
Sweden

Elaine J. Weyuker  
Mälardalen University  
Sweden  
University of Central Florida  
USA

Daniel Sundmark  
Mälardalen University  
Sweden

## ABSTRACT

Software testing is sometimes plagued with intermittently failing tests and finding the root causes of such failing tests is often difficult. This problem has been widely studied at the unit testing level for open source software, but there has been far less investigation at the system test level, particularly the testing of industrial embedded systems. This paper describes our investigation of the root causes of intermittently failing tests in the embedded systems domain, with the goal of better understanding, explaining and categorizing the underlying faults. The subject of our investigation is a currently-running industrial embedded system, along with the system level testing that was performed. We devised and used a novel metric for classifying test cases as intermittent. From more than a half million test verdicts, we identified intermittently and consistently failing tests, and identified their root causes using multiple sources. We found that about 1-3% of all test cases were intermittently failing. From analysis of the case study results and related work, we identified nine factors associated with test case intermittence. We found that a fix for a consistently failing test typically removed a larger number of failures detected by other tests than a fix for an intermittent test. We also found that more effort was usually needed to identify fixes for intermittent tests than for consistent tests. An overlap between root causes leading to intermittent and consistent tests was identified. Many root causes of intermittence are the same in industrial embedded systems and open source software. However, when comparing unit testing to system level testing, especially for embedded systems, we observed that the test environment itself is often the cause of intermittence.

## CCS CONCEPTS

- Software and its engineering → Software creation and management; Software testing and debugging; Empirical software validation.

## KEYWORDS

system level test automation, embedded systems, flaky tests, intermittently failing tests, non-deterministic tests

### ACM Reference Format:

Per Erik Strandberg, Thomas J. Ostrand, Elaine J. Weyuker, Wasif Afzal, and Daniel Sundmark. 2020. Intermittently Failing Tests in the Embedded Systems Domain. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397359>

## 1 INTRODUCTION

The software in embedded systems has to be tested under realistic conditions, using real hardware [8, 47]. Even when using continuous practices, such as nightly testing, system testing is a resource constrained process when compared to unit level testing, and there is typically not enough time to execute all test cases every night on all hardware versions [31, 41]. If test cases fail intermittently in nightly testing, troubleshooting is often very costly and reproduction of failures very difficult, leading staff to distrust test results [29].

Intermittently failing hardware devices have been studied for at least 70 years [12]. As software-based functionality has become increasingly important in large systems, non-deterministic results have continued to be a problem [7, 28, 30]. With improved automation and continuous integration, the volume of test results exacerbates the problem [36, 39]. This problem has been studied in related work as described in Section 2, focusing primarily on unit testing level of open source software. The conclusion has generally been that test cases of poor quality fail unpredictably, even when testing the same software, and that the test cases are to blame for intermittent failures [29, 45]. This paper continues this research, but focuses on industrial *embedded systems* being developed and tested at the *system level* with evolving software and testware. In this type of environment we hypothesize that intermittently failing tests may have root causes that can stem from the testware, hardware, software, or their interfaces.

The **research objectives** of this paper are to identify intermittently failing tests during system level testing by using an easily computed measure of the frequency of change of test verdict. Additionally we want to find, explain and categorize the root causes of such failures in the context of development and maintenance of industrial embedded systems designed by using continuous integration, and to compare our findings with similar research to build knowledge in the topic.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397359>

The **main contributions** of this paper are:

- (1) Nine factors that may lead to intermittently failing tests for system level testing of an embedded system: test case assumptions, complexity of testing, software or hardware faults, test case dependencies, resource leaks, network issues, random number issues, test system issues, and code maintenance.
- (2) Definition of a metric that can identify intermittently failing tests, and can measure the frequency of intermittence over the test base of a system.
- (3) Evidence that failures detected by intermittently failing tests often require more work to find their root cause than those detected by consistently failing tests. This may lead testers to give up without identifying the failure's root cause, and to rely on the hope that failures will occur infrequently enough that they can be ignored.
- (4) Experience that a fix for a consistently failing test often repairs a larger number of failures found by other test cases than a fix of an intermittently failing test.

The paper is organized as follows. Section 2.1 presents terminology. Section 2.2 describes two real cases of intermittent tests that motivate our research. Section 2.3 introduces a novel metric for measuring intermittence, which is then used to analyze data collected in the case study presented in Section 3. Section 4 presents the results of the case study, and Section 5 shows how our results differ from, confirm, and extend previous work. Sections 6 and 8 discuss our findings and our conclusions.

## 2 INTERMITTENTLY FAILING TESTS

### 2.1 Terminology

In addition to the obvious hardware (HW) and software (SW) that comprise an embedded system (ES), development and testing also involve *testware* (TW), which itself has both software and hardware components. The TW software includes items such as test harnesses and stubs, servers, test libraries, test scripts, automated test cases and code to control the automation. TW hardware is the physical environment on which test cases are executed, including servers, cables, and peripheral equipment such as load generators.

The hoped-for result of running a regression test suite is that all tests in the suite will be successful, thus confirming that recent modifications to the system have not broken any existing functionality or property. With *continuous integration* (CI) methods, changes may occur at any time, and regression testing may be carried out daily, or even more frequently, to assure that recent changes have not adversely affected the system's behavior.

During the development and maintenance of a large industrial embedded system, we have observed many examples of test cases that produce different verdicts in successive regression testing runs, with sequences of mixed verdicts. Such results occur even for test cases that have no apparent connection to system modifications that were made since the previous test run.

*Flaky tests* are tests that yield differing verdicts when nothing in the SW, HW or TW have been changed [29]. The different verdicts occur because of hidden changes in the system state or the application's environment. State changes can be caused by previously run test cases or by normal operation of the system. Environment

changes can occur spontaneously, and cause problems when system designers have failed to consider the possibility of their occurrence. Tests can also be flaky because of poor design. These are sometimes called *smelly tests* [18, 21].

In many industrial contexts, including the one that is the subject of this case study, it is not useful to look for flaky tests, due to frequent changes made in the underlying SW, HW and TW in a CI development paradigm. Since the perceived business value of retesting is limited when nothing has changed, we cannot expect to see much repeated testing on unchanged SW, TW and HW. Thus, although potentially flaky tests may exist, they are rarely observed and in fact represent a lack of understanding of all of the factors that might impact system behavior. Bell et al. [9] developed a tool to detect flaky tests without retesting that relies on instrumentation (code coverage), but this may be hard or impossible to use for CI, or for testing SW in resource constrained embedded systems [15].

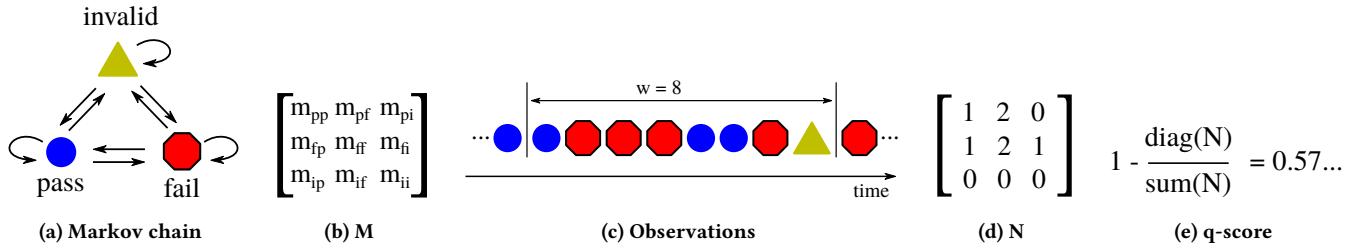
For these reasons, we define an *intermittently failing test* to be a test case that has been executed repeatedly while there is a potential evolution in SW, HW or TW, and where the verdict changes over time. Note that both flaky and intermittently failing tests refer to the dynamic execution of test cases, and require that the system be executed at least twice before we can label them as such. Furthermore, in automated system level testing of ES, intermittently failing tests are different from flaky tests in that they allow changes in the SW or HW of the ES under test, as well as in the TW used for testing.

A reader might object to our definition and argue that when changing an ES one should not be surprised that verdicts from test cases change as a result. Although this is of course a relevant comment, in this paper we do not distinguish between expected and unexpected intermittently failing tests – we are interested in these tests regardless of root cause. A reader might also object to test cases being “the same” test cases, if the underlying test script code has been modified. This is also a relevant comment, and as we will see in Section 5, code maintenance is indeed a factor that impacts intermittence, but not a dominating factor.

### 2.2 Two Examples

2.2.1 *Example 1: A Smelly, Flaky and Intermittently Failing Test.* One test case at the company targets the temperature sensor that measures the internal temperature of the HW, and reports the value to other software. If the SW receives a plausible value, the test case assumes that the sensor is working properly.

The test was designed around the assumption that the ambient temperature would be between 20 and 40°C with an acceptable internal temperature between 21 and 60°C. This assumption was valid for several years and the test hardly ever failed. However, at one time the test lab was moved to a different location, and the test started to fail intermittently on one of the test systems. The new test lab had much better air conditioning, with one particular test system placed very close to one of the air conditioners. If HW in this test system had been powered off for some time, allowing it to cool down, and if the temperature sensor test was early in the test suite, prior to when the HW had heated up, then the test would fail because the internal temperature reported by the sensor was outside the plausible range. Since these two conditions rarely



**Figure 1: We model the test cases as Markov chains. The test case has some unknown transition probabilities, shown in matrix  $M$ . Counts of the 7 observed transitions in a window of length 8 are in matrix  $N$ , and the observed intermittence is computed with the  $q$ -score. Details described in Section 2.3.**

occurred simultaneously, the failure was rarely observed. After a discussion with HW engineers, the plausible range was changed to 15 to 50°C, and the test stopped failing.

One could argue that this test case was smelly, because of the false underlying assumption of what an acceptable temperature range would be. The test case was also intermittently failing. However, in order to truly be able to label the test as flaky, we would have to investigate whether or not the test case had been executed at least twice with the same SW, HW and TW combination while also producing different verdicts.

The temperature sensor test shows an almost trivial example of a fix, but it still illustrates the effort needed in identifying and dealing with intermittently failing tests: first, test results have to be observed, discussed and understood. Then an informed fix has to be applied – a prerequisite for this particular fix was a discussion across roles (test engineer and HW engineer). Once the fix is applied, the test results have to be observed again for some time, to see that the issue has been properly addressed and there are no more failures.

This example also illustrates that “the system state” when testing an ES at system level can be very complex: the state contains not only all variables of the Linux kernel, the many physical components and possible flaws in soldering of the device, the test scripts and the order in which test cases are executed. In this particular example, the system state also includes room temperature, and the internal temperature of the devices. These parameters were not fully considered when the test script was first written. Sometimes, we cannot observe or control all variables in the system state, because we do not even know all of the factors that might impact the system’s behavior.

**2.2.2 Example 2: Resource Leak in SW.** Another example further illustrates the complexity of automated testing of ES at the system level. In one of the features in the SW under study, there was a bug in the form of a resource leak. This feature was exercised by several test cases that could all trigger the leak. But in order to observe the bug, a certain number of the test cases had to be executed for the resource to be depleted. At the company, there is a regression test selection system in place that generates new test suites every night, and one of the factors that leads to an increased priority of a test case is that it has failed recently [37, 41]. Time for testing is a scarce resource, and testing terminates once time has run out, potentially leaving test cases that are late in the suite unexecuted. Therefore, if a certain test case passed recently, it might have a

relatively low priority and not be executed at all during nightly testing. But if the test case did fail, thereby identifying the bug, it would be prioritized and placed early in the suite. However, in the beginning of the suite it was unlikely that the leak had been triggered enough for the test to fail. In retrospect, we know that the leak was consistent, but because test suites changed over time, we did not consistently trigger the SW bug. We thus observed an intermittently failing test case where we could have observed a consistently failing test case if the suites had been identical over time. This is a very common issue when there are resource leakages, and one of the reasons that these sorts of faults are so difficult to find and fix properly. Essentially, the leakage will only occur over a long period of time or when an unusual surge in workload has occurred. If nightly testing begins in a standard or quiescent state, it might be that the failure is observed very rarely and only when a particular test case order occurs.

### 2.3 A Model for Intermittently Failing Tests

We want to define a property of test cases that captures the likelihood of a test failing intermittently, regardless of reason. To model the sequence of varying test case results, we use the concept of a *Markov chain* [34], which is a process with well-defined states, and probabilities for transitions from one state to another that depend only on the current state and input. When run, test cases may result in one of three verdicts: pass, fail or invalid, where invalid means that the execution could not be completed. This is typically triggered by unexpected behavior in the HW, SW or TW, followed by an unhandled exception in the TW. We can think of the test case verdicts (pass, fail, invalid) as being states in a Markov chain (illustrated in Figure 1a), and each execution of the test case as a transition. When a test verdict is different from the previous one, the transition is to a different state; a repeated verdict is represented by a transition to the same state. Each test case has some unknown probabilities for the transitions, as shown in a transition matrix  $M$  (see Fig. 1b). The probabilities may change over time because of changes in the SW, TW, and HW, and in practice we do not know what they are.

Based on historic data, we can observe the actual verdicts that have been produced (Fig. 1c). By counting transitions that occur in a sequence of consecutive test case executions, we construct the matrix  $N$ , in which each element represents the number of observed transitions from state to state (Fig. 1d). While the elements  $m_{i,j}$  of

$M$  are the unknown *probabilities* of going from  $i$  to  $j$ , the  $n_{i,j}$  are the *number of observed transitions* from  $i$  to  $j$ , and are dependent on the sequence of test cases actually run. For example,  $n_{p,f}$  is the number of observed transitions from pass to fail, corresponding to the number of times that a pass verdict is immediately followed by a fail verdict. The diagonal elements of  $N$  represent how often the test case verdict has remained the same, and the off-diagonal elements represent how often the test case verdict has changed, i.e., how often the Markov chain changes state. We now define the *q-score* as the fraction of results in which the test case changes state, i.e.  $q = 1 - \text{diag}(N)/\text{sum}(N)$ , (Fig. 1e).

The q-score is a direct measure of the variability of a test case's verdicts. We have found it to be most useful when continuously measured over a moving window of approximately 8-15 consecutive verdicts of a test case. A running plot of the score shows the varying intermittence over time of the test case, and shows whether the intermittence is increasing or decreasing. Measurement over long periods such as several months can be used to identify test cases that have intermittency somewhere in their history, but is not useful for determining their most recent status.

Figure 1 illustrates a window size of eight, giving seven observed transitions. The test case remains in the same state three times (one pass → pass, two fail → fail) and changes state four times, indicating that the test case is more likely to change verdict than to retain the verdict. The computed q-score for this window is 0.57. Given a threshold value of the q-score, one could now use this metric as a method for detecting intermittently failing tests. We can also compare the observed intermittence of test cases, e.g., a test with a q-score of 0.57 could be thought of as more intermittent than a test case with a q-score of 0.15. Thus rather than simply deciding that a test case is either intermittently failing or not, much as the literature on flaky tests has done, we can instead use the q-score as a way of deciding which tests are most problematic because of their changing behavior. Of course, it may be that test cases exhibit a high q-score for totally expected reasons such as when the SW or test scripts are being refactored, or undergoing code maintenance or when the HW is being replaced. One could therefore consider combining the q-score with other metrics, such as code churn.

It is also useful to introduce a metric for how frequently a test case passes. We define the p-score as the fraction of passed verdicts in a window of observed verdicts. Using the same example as above (Figure 1),  $p = 3/8 = 0.375$ , indicating that this test case passed in 37.5% of the executions.

Both the windowed p-score and q-score will be used to select test cases of interest from a large pool of possible test cases in the coming sections. We will also use the overall scores from all verdicts for each test case to describe the intermittence of test cases on a population level.

As we will demonstrate in our case study, different q-score thresholds and window sizes will select different test cases as potentially problematic. The q-score's range of values from 0 to 1 provides the system tester a more precise means of setting the degree of variability to indicate which test case results are most urgent to investigate, in contrast to a binary metric such as flaky tests.

### 3 CASE STUDY DESIGN

This section describes the research questions, the case, and the industrial context, as well as data collection, and data analysis procedures. The overall research flow is illustrated in Figure 2.

Our goal is to explore and explain the root causes of intermittently failing tests observed during system level testing in the development and maintenance of embedded systems developed using continuous integration in an industrial environment. We will compare these test cases with those that consistently cause failures. We formulate two research questions (RQs):

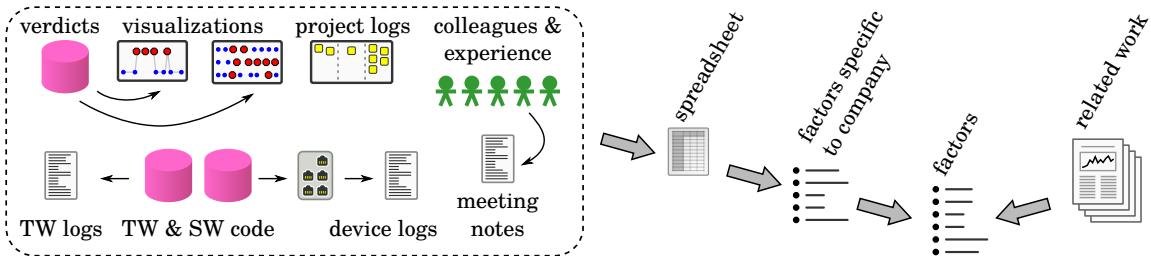
- **RQ1:** What are the root causes of intermittently failing tests when testing embedded systems on a system level?
- **RQ2:** Are the root causes different for tests that fail consistently?

#### 3.1 Case and Subject Selection

The *case* in this study is nine months of test results from nightly testing of an operating system for embedded systems. We analyze the case using four units of analysis: two groups of intermittently failing tests, and two groups of consistently failing tests. This study could thus be defined as an embedded exploratory and explanatory case study with four units of analysis [35].

**3.1.1 Industrial Context.** Recent research literature on flaky tests has typically focused on unit testing of open source projects targeting a general-purpose computer (not ES). To broaden our understanding of intermittently failing tests in industrial systems, the case study is conducted at Westermo Network Technologies AB an international company with about 250 employees targeting the on-board rail, track-side rail, power distribution, and other industrial automation domains where communication networks in harsh environments are needed. The company develops switches and routers (the HW) running an operating system (the SW), which is developed in-house in an agile development process based on Kanban where new features are developed in separate code branches. The company has invested heavily in test automation. The testware (TW) includes an internal infrastructure for nightly testing, equipment such as test servers and devices under test with a combined weight of more than a ton, and a test framework with hundreds of test scripts, to which new ones are added weekly. Several person-years of effort have been invested in coding and construction of the TW. SW development takes place in feature branches that are expected to receive many code changes, and once a feature is implemented and stabilized, it is merged into a master branch.

To get an understanding of how frequently the same SW is tested on consecutive nights (if at all), we analyzed how many nights in a row the same SW, the same TW, and both the same SW and TW are used. The median number of nights of regression testing with the same revision of the SW was 2.5 in the branch we investigated. The corresponding number for TW was only 1 (see details in Table 1). In addition to code changes, there are many variables in the TW that are hard to observe, log, or control: network latency, room temperature where devices operate, states in test servers, etc. When testing an embedded system at the system level, running only a subset of the regression test suite may be necessary, because testing takes far more time at the system level than testing done at the



**Figure 2:** Using several sources of information, we collected information on root causes and fixes for intermittently failing tests. With related work we created a list of relevant factors.

**Table 1: Number of consecutive nights of testing with the same SW, TW, or both.**

Nights w. same	Min	Max	Avg.	Med.	Std.d.
SW Code	1	107	5.72	2.5	13.97
TW Code	1	28	3.02	1.0	3.96
SW and TW Code	1	17	2.32	1.0	2.68

unit level. Selecting a subset will typically lead to test cases being reordered in the test suites [37, 41], which may trigger intermittent failures if there are test case dependencies.

The software quality process at the company includes several types of testing techniques in addition to automated regression testing (such as manual risk-based testing, robustness testing, etc.), as well as bug tracking, bug triage meetings, test results sync meetings, and release gate meetings, before a new version of the SW is made available to customers.

### 3.2 Data Collection Procedures

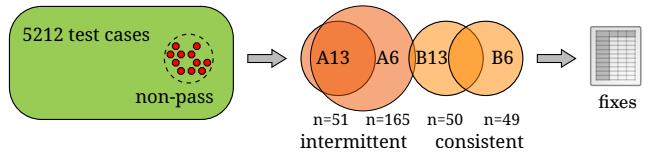
**3.2.1 Raw Data.** After discussions with Westermo, we extracted verdicts from 270 consecutive nights of testing of a stable code branch. This branch has existed for a long time, so we expected to see mostly passing tests. By using a stable code branch, the impact of changes in SW should be minimized, in contrast to a feature branch that is short-lived and has frequent SW changes. In such a case we would expect many failing tests. The extracted data contains 532069 verdicts from 5212 test cases. In this study, we consider a test case to be a combination of one of the 13 test systems, 527 test scripts, and 69 different parameter settings in the raw data. Note that if the same script is run on two different systems, we count it as two different test cases.

Each individual test case ran at most once each night, so had between 1 and 270 verdicts during this period. One test case is hard coded into the beginning of the scripts, so it ran every night. The average was 102.1 verdicts with a median of 96 and standard deviation of 40.3. This means that each night about a third of all test cases were included in nightly testing of this branch, with the particular test cases selected varying from night to night.

Of the 532069 verdicts, 526335 (98.9%) were pass, 2673 (0.5%) were fail, and 3061 (0.6%) invalid. For this study, we are not concerned with the impact of the failed and invalid verdicts, so there is no severity associated with the non-pass verdicts.

**Table 2: Minimum, maximum, average, median and standard deviation of the q-scores and p-scores of all 5205 test cases that were executed more than once.**

Score	Min	Max	Avg.	Med.	Std.d.
p-score	0.0	1.00	0.986	1.0	0.060
q-score	0.0	0.68	0.011	0.0	0.037



**Figure 3:** From all test cases we identified intermittent and consistent tests, and analyzed their fixes.

We removed the seven test cases that had only been executed once during the 270 night period. Based on the complete sequence of verdicts for each remaining test case, we found that the tests have an average p-score of 0.986 and an average q-score of 0.011. This means that the test cases could be thought of as having an overall probability of 1.1% to change verdict in this data set. The maximum q-score of 0.68 means that at least one or a few test cases were more likely to change verdict than to retain a verdict. The median p-score of 1.0 and q-score of 0.0 indicates that almost all test cases never fail, and never change verdict. 24.8% of the test cases had a non-zero q-score. See details in Table 2.

**3.2.2 Categorizing Intermittently and Consistently Failing Tests.** Our goal is to identify and study fixes such that we can determine the root causes of intermittently and consistently failing tests. From the 5212 tests in the database, we identified the following two groups of tests:

- Group A: test cases that at some point in time intermittently failed (i.e. had a high q-score), but after fixes were made, then mostly passed (i.e. had a high p-score).
- Group B: test cases that at some point in time failed consistently (i.e. had a low p-score and also low q-score), but after changes were made, they mostly passed (i.e. had a high p-score).

To study changes in q-score and p-score over time we use moving windows of verdicts, as illustrated in Figure 1. By using a small

window we hope to capture rapid changes in q-score, and with a larger window we hope to capture slower trends. In general, we want the A and B groups to contain enough test cases to permit meaningful analysis, while not too many to be overwhelming. We generated many samples of the A and B groups, using window sizes ranging from 5 to 30 and different cut-off values of p and q. In the end, we used two window sizes for a total of four groups – groups A6 and B6 used a window size of 6 verdicts, while A13 and B13 used 13 verdicts. For all four groups we required that they ended up mostly passing, with a final p-score of at least of 0.96. For group A13, we required a q-score of at least 0.35 at some point in time, and for A6, we required a q-score of at least 0.5. For B6 and B13, we required the p-score to go below 0.2 at some point in time. With these window sizes and thresholds, we got a total of 230 test cases in the four groups.

To summarize, for inclusion in A, a test case must: (i) at some point, have had a floating q-score over a certain limit (0.5 for A6, and 0.35 for A13) and (ii) the final floating p-score must end above a certain limit (0.96 for both A6 and A13). These are thus test cases that have been intermittently failing, but end up passing (examples are visualized in Figures 4a and 4b). For inclusion in B, a test case must: (i) not be in A<sup>1</sup>, (ii) at some point have had a p-score below a certain limit (0.2 for both B6 and B13), and (iii) the final floating p-score must end above another p-score limit (0.96 for both B6 and B13). In other words, these are test cases that have been mostly failing for a period, but that end up passing (an example is visualized in Fig. 4c). This resulted in a total of 230 test cases, with 165 test cases in A6, 51 in A13, 49 in B6 and 50 in B13. Figure 3 shows the large overlap between the test groups, as 49 of the 51 tests in group A13 were also present in A6, 34 of the 49 tests in B6 were also present in B13, and two of the test cases were in both A6 and B13. No test case was in three or four groups.

### 3.3 Analysis Procedure

The 230 test cases of interest were analyzed. Typically this investigation required more than one source of information and involved inspection of test and project artifacts, or visualizations thereof. The tools used were: (i) A visualization of the verdicts from the test case, as illustrated in Figure 4. (ii) A second type of visualization, this one heatmap-like, that uses test results from before or after the date range we investigated. These plots can quickly determine whether the test case is still under investigation, i.e., if a test case was failing in a later phase and no obvious fix has been found, then it was assumed that it was still under investigation. Example visualizations of the heatmap plot used can be found in Figure 10 in [38]. In addition to these visualizations, (iii) the SQL-interface to the test results database was also used for answering non-standard queries, such as at what time a certain piece of HW was replaced in a test system. Log files from (iv) the test framework, as well as test framework communication logs with (v) devices under test and (vi) peripheral equipment such as load generators in the test systems were manually inspected for error messages or other issues. Project artifacts in (vii) issue trackers and (viii) planning tools were also

used, as well as (ix) historic hand written notes from developer-tester sync meetings from the time range of investigation. In some cases, the source code repository logs of (x) the test framework code or (xi) SW code were investigated. In addition to these tools, scripts and artifacts, we also used (xii) personal experience of Westermo staff (including the first author).

For each of the 230 test cases, observations were hand-written on paper, resulting in 34 pages by the end of the study. An anonymized summary was written in an on-line spreadsheet for simplified collaboration between authors. For each test case we made notes on tools used, fix identified or root cause. A typical example of the written notes is:

- **Tools used:** test framework log file, heatmap, test framework source code changes.
- **Fix:** Revert of code change that applied the system configuration in incomplete steps, instead of a complete and correct configuration.
- **Root Cause:** Code change in test script that applies an intermediate (broken) configuration involving three features.

When all 230 test cases had been analyzed, we grouped the fixes or root causes into categories and subcategories (summarized in Table 3 and discussed in Section 4).

## 4 CASE STUDY RESULTS

### 4.1 Fixes and Root Causes (RQ1)

We identified five main categories of root causes: (i) Changed HW allocation for testing, (ii) test case assumptions, (iii) test system issues, (iv) SW or HW Faults, and (v) code maintenance of TW code. In the analysis, we also identified test cases still under investigation, test cases that had more than one fix or root cause, and test cases for which we were unable to identify a fix or root cause. Table 3 summarizes these categories. In the following paragraphs, we discuss each category in detail.

**Hardware Resource Allocation:** Seven fixes involved modifications of the HW allocation. This means testing with a different subset of the available physical devices, but with the same test script on the same test system (the HW selection algorithm is presented in [40]). Five of the fixes involved avoiding a link breaker that was used incorrectly (or not at all) by the test, but that could still interfere with traffic. One test script required a port with nothing on the other end, and another required traffic to flow through the back plane of the device in ways not adhered to. In total, the seven fixes stabilized nine combinations of test script, parameter settings, and test system.

**Test Case Assumptions:** Nine fixes repaired incorrect assumptions about the test framework and test systems, particularly assumptions involving timing (5 of 9). In addition to changed intervals or tolerances on timing, one test script required a modified temperature range (as mentioned in Section 2.2.1). One fix involved a search for an event in the latest logged events in a log file in the HW, but due to other activities using the same log file, the test script had to expand the search to a larger portion of the file. Another test script assumed that exactly one instance of a unique type of HW was present in the test system, and could fail if more than one was present. Yet another fix was to update the TW code that generates traffic. The final fix in this category solved assumptions on the

<sup>1</sup>For inclusion in B, we applied the criteria of not being in A for a given window size, e.g. we allowed inclusion of a test case in both A6 and B13, but not in A6 and B6.

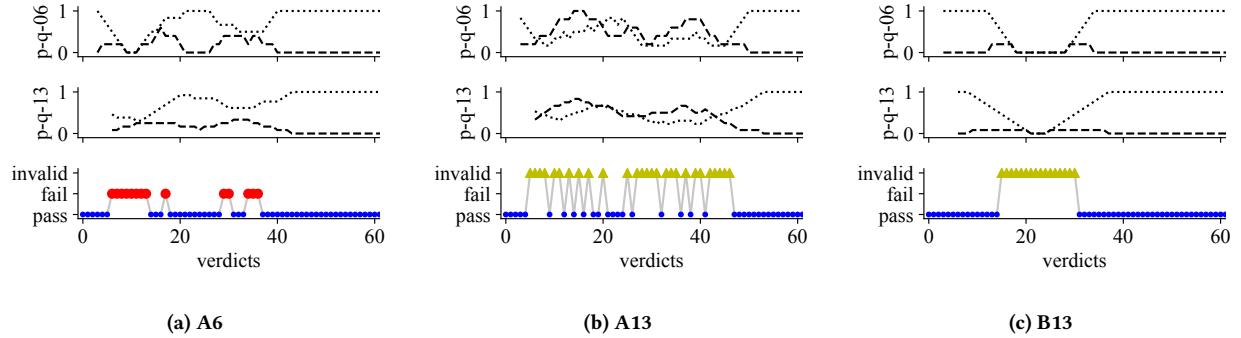


Figure 4: Verdicts over time from three test cases, as well as q-score (dashed) and p-score (dotted) for window size 6 and 13.

Table 3: Distinct categories of fixes. E.g., there was a total of 5 distinct test cases with HW allocation of linkbreaker as root cause, 4 of which were in A6, 4 were in A13 and 1 in B6.

Root Cause/Fix	A6	A13	B6	B13	Tot.
HW Allocation	4	4	3	0	7
- link breaker	4	4	1	0	5
- switch core	0	0	1	0	1
- empty port	0	0	1	0	1
TC Assumptions	7	3	2	1	9
- timing	5	2	0	0	5
- test system layout	1	1	0	0	1
- temperature	1	0	0	0	1
- log file	0	0	1	0	1
- lib. version	0	0	1	1	1
Test System Issues	9	6	2	2	11
- replace device	4	3	0	0	4
- console junk	2	1	0	0	2
- I/O relay	2	2	0	0	2
- USB sticks	1	0	0	1	1
- FTP server	0	0	1	1	1
- license	0	0	1	0	1
SW or HW Faults	16	2	0	0	16
- SW impact on HW	7	1	0	0	7
- SW timing	9	1	0	0	9
Code Maintenance	9	0	2	3	12
- unclear	7	0	0	1	8
- broken renaming	1	0	1	1	2
- traffic generator	1	0	0	0	1
- forgotten patch	0	0	1	1	1
Multiple Root Causes	8	2	0	1	8
Under Investigation	14	34	0	0	35
Unknown Fix	8	5	0	0	8

**Test System Issues:** Eleven fixes were modifications to the test systems: four of these involved replacing devices.<sup>2</sup> Two were fixed by rebooting a server when junk characters were sometimes written in the console used between the TW and the HW. In the test systems at Westermo, relays are used to power on and off devices. Two of the fixes in A6 and A13 were related to worn out relays that had to be replaced (newer test systems use solid state relays to avoid this problem). One fix was to complete the configuration of an FTP server that had not been finalized on a new test system. Another fix was to insert USB-sticks that were assumed to be present in the HW, but had been forgotten, lost or broken. Finally, one fix was related to a communication protocol that requires a license, but after replacing HW in the test system, the license was no longer valid. In total, the 11 fixes stabilized 18 combinations of test script, parameter settings, and test system.

**SW or HW Faults:** 16 root causes were related to faults in the SW or HW. Nine had a root cause in timing issues in the SW leading to intermittent failures, and seven were related to intermittent failures due to root causes in SW making a port in the HW temporarily unusable and thereby blocking traffic. None of the SW or HW faults led to consistent failures – apparently, any such fault had already been taken care of in features branches, i.e., these faults never reached the stable code branch we collected data from. With duplicates, these 16 root causes accounted for a total of 53 combinations of test script, parameter settings, and test systems.

**Code Maintenance:** A total of 49 test script, parameter, and test system combinations were related to refactoring or maintenance of TW code, with a total of 12 distinct fixes. Two fixes had a root cause in renamed variables where not all instances of their use had been identified, which would lead to attribute errors at run time. Another fix was related to a new feature being developed in a separate feature branch of the SW. Once the feature was completed, several test cases were modified to account for some changes in behavior of the SW. However, not all test scripts that used this feature had been identified and modified. The remaining root causes coincided with maintenance or refactoring of the TW code, e.g., if many code changes had been added to a test case that failed intermittently, it was assumed to be undergoing maintenance.

<sup>2</sup>When developing new HW products, Westermo runs prototypes in nightly testing to verify SW-HW integration. Over time, one test system might use several generations of prototypes, as well as released HW products.

versions of a third party library used by the test framework. In total, the 9 fixes stabilized 31 combinations of test script, parameter settings, and test system.

**Under Investigation:** 35 root causes (and a total of 53 combinations) were still under investigation, almost all of which were related to A6. Two identified root causes seemed to be related to other intermittent problems in the TW, and TW assumptions on HW performance leading to undesired reboots of HW.

**Unknown and Overlapping Root Causes:** For eight test cases, we found no obvious fix (with 9 total combinations of test script, parameters, and test system). Another eight had two or more overlapping root causes (total: 8 combinations).

**Duplicated Fixes:** If two test cases were fixed by the same fix or seemed to have the same root cause, they were considered duplicates, but if similar fixes had to be applied several times, we treated them as separate fixes. More than half of the combinations of test script, parameter settings, and test systems were duplicates (124 of 230). The most duplicated subcategories were SW or HW faults, under investigation, code maintenance, and test case assumptions; only 10 of the 124 duplicates belonged to other sub-categories.

## 4.2 Differences Between Intermittent and Consistent Tests (RQ2)

The most important differences we observed between (A) the intermittently failing tests, and (B) the consistently failing tests, were that the B-group contained no test case with unknown root causes, no test case with SW or HW faults, and no test cases that were still under investigation. On the other hand, the A-groups had several test cases in these categories. This provided evidence that confirmed our intuition that intermittency often made fault diagnosis significantly more difficult. Another difference is that the B groups have a larger number of duplicates: group B6 had 9 non-duplicates and 40 duplicates, meaning that 9 fixes would resolve all 49 non-passing test cases in B6. In contrast, group A13 had only about 30% duplicates, meaning that a fix for an individual test case was not very likely to also fix another test script, parameter, and test system combination.

The different fixes and root causes required different amounts of effort to identify, using the tools explained in Section 3.3. On average, we used 1.1 tools during the investigation when finding a duplicate, and 2.9 tools when finding a SW or HW fault. During the investigation, unknown fixes required 4.5 tools, and the remaining six categories between 2.2 and 2.6 tools. The average number of tools required was 2.0 for both A6 and A13, and 1.2 for both B6 and B13. This supports the hypothesis that a greater effort is needed to find the root cause of an intermittently failing test.

## 5 RQ1 REVISITED IN LIGHT OF RELATED WORK

Despite starting with a data set including half a million verdicts, our findings obviously do not provide a complete picture of intermittently failing tests in the embedded systems domain, since we only investigated a single system at a single company. In this section, we revisit RQ1 and analyze our findings with respect to previous research. In particular, we have explored literature on intermittence in embedded and electrical systems, including a 70 year old paper, literature on how embedded systems are tested, as well as recent literature on flaky tests.

We have found only four papers in which intermittently failing tests have been investigated at the system level: Ahmad et al. [3], Eck et al. [14], Lam et al. [27], and Thorve et al. [43]. These papers report on intermittently failing tests in industrial embedded systems, Mozilla programs in various operating systems, Microsoft programs, and Android apps. In particular, they report that intermittently failing tests are not uncommon, and that they represent a real problem. Based on our findings in the current case study, these four papers and other related work, we made some general observations, and identified nine factors relevant for intermittently failing tests in the embedded systems domain.

**General Observations:** The term *flaky tests* for the domain of regression testing and continuous integration was popularized in blog posts, e.g. [17]. Luo et al. [29] did an investigation on unit tests in open source projects. They found that many tests were flaky because of asynchronous waiting, concurrency, or test order. Important problems with flaky tests are they can be hard to reproduce, may waste time/involve maintenance cost, they can hide other bugs and they can reduce the confidence in testing such that practitioners ignore failing tests [29, 44]. The findings of our case study support the generalization of many of these findings to the system level of embedded systems. We saw that finding the root cause for intermittently failing tests is frequently more difficult than for consistently failing tests, and that one fix for a consistently failing test typically repairs several other issues, whereas a fix of an intermittently failing test repairs fewer others. We also saw that some test cases could have overlapping root causes, i.e., they were intermittently failing for more than one reason, which supports the idea that intermittently failing tests may hide bugs.

**Measuring Intermittence:** The q-score metric is similar to the model presented by Breuer in 1973 [10], but discovered independently, and created for another domain (regression testing of SW-intense embedded systems as opposed to describing faults in HW). As far as we can tell, Breuer was first to use Markov chains to describe faults in a system (as opposed to modeling a system under test). Another way of quantifying flakiness, based on entropy, was presented by Gao [19]. This method requires code instrumentation (code coverage), and was evaluated using Java programs with 9 to 90 thousand lines of code. One perceived advantage of Gao's metric is that it can be used to "weed out flaky failures." Labuschagne et al. [26] investigated build failures in continuous integration environments of open source software. As part of their data analysis, they used a Markov chain similar to ours. They investigated open source software when a "build" (a test suite) fails, and their use of transitions between build states was used to identify when exactly one non-pass build had occurred.

**Factor 1: Test Case Assumptions.** Test case assumptions include issues such as poor ranges, tolerances, timing, or concurrency. This was noted by several papers on system level testing (e.g., [14, 27, 43]) and was also seen in our case study. Abbaspour Asadollah et al. [2] investigated concurrency bugs in the domain of open source SW and found that about 4% of bugs were related to concurrency, and that they were slightly more severe and required slightly more time to fix than other bugs. Musuvathi et al. [32] showed that some issues in concurrent programs could consistently be reproduced if threading was monitored and controlled.

**Factor 2: Complexity of Testing.** As illustrated in the example in Section 2.2.2, testing an embedded systems can be a complex process, in particular when compared to running unit tests without target HW. Testing of embedded systems may require HW testing, extra-functional properties testing, network testing, system testing, test execution on one or more systems under test, systematic and exploratory test case design, etc. [1, 20]. Eldh et al. [16] analyzed faults in a telecom SW and found that many faults are not discovered in unit level testing because of complexity in testing that developers do not always understand, and that many of the reported faults were not related to SW. Furthermore, Ostrand and Weyuker [33] found that different types of faults were uncovered during unit testing than were found during system testing of an industrial software system, concluding that both types of testing were essential.

Regardless of whether or not the complexity of the TW is a *root cause* for intermittently failing tests, it can be an exacerbating factor. It was noted as such by [3, 14, 43], as well as in our case study under the HW allocation and test system issues categories. Complexity is a challenge for testing an embedded systems in general, not only for intermittently failing tests [39]. To mitigate the complexity, Lam et al. [27] indicate that the use of log file analysis may be helpful, Ahmad et al. [3] indicate that improved test results reporting may be helpful, and our previous work draws similar conclusions for testing of embedded systems in general [38]. Similarly, Jiang et al. [25] used text mining and comparisons of industrial test execution logs from system and integration level testing at Huawei-Tech Inc. to reduce the burden of determining the root cause of failing tests. The method was successfully deployed and used in industry. Furthermore, Herzig and Nagappan [24] analyzed patterns in test cases at Microsoft to predict if a failed test case had a root cause in TW, with the goal of reducing development effort.

Mårtensson et al. [31] investigated problems and experiences when striving for continuous integration of embedded systems. They found that the test environment is a limited resource, leading to a tendency to construct many test systems with custom HW. This in turn may lead to tests being intermittent because of the increased maintenance burden. They also observe that test cases that pass on simulated HW in a local build, are not guaranteed to pass when tested on real HW. Wiklund et al. [46] came to similar conclusions for test automation in general.

**Factor 3: SW or HW Faults.** The challenge of intermittent faults in electrical systems has been known for a long time. The earliest study we have been able to find on the topic is from 1947 in which Cooper [12] investigated electrical control of dangerous machines. In the 1960s, Ball and Hardie [7] published a paper describing intermittent failures. The authors write that their “experience has shown that field failures ... tend to be intermittent in nature.” A few years later, Breuer [10] investigated testing of intermittent faults in digital circuits.

A forty year old paper by Malaiya and Su [30] investigated intermittent faults in integrated circuits. A similar study was made more recently by Bakhshi et al. [6]. Both Malaiya and Su, and Bakhshi et al. came to similar conclusions regarding intermittent faults caused by HW: temperature, loose connectors, bad soldering, corrosion, and voltage fluctuations. Bakhshi et al. also mentions possible root causes caused by SW, such as timing failures, processor loads, memory leaks, and disk error.

Some bugs, sometimes referred to as Mandelbugs [22], cannot be observed without state build-up. This causes a possible delay between activation and symptom, may require non-trivial timing or a particular sequence for activation, or have a non-trivial error-propagation. The root causes identified in the ‘SW or HW faults’ factor of our case study could be described as Mandelbugs. According to Cavezza et al. [11], a large portion of faults in both mission-critical SW and open source SW are due to Mandelbugs. Di Martino et al. [13] investigated repair logs from a super computer and found that it was not uncommon for SW faults to propagate from one node to another, whereas this was rare for HW faults. They also found that HW faults were more numerous than SW faults, but that HW faults were more rapidly repaired than SW faults. Sycofyllos found that most SW-related fatal failures, regardless of domain, have a root cause involving not only SW, but *both* SW and a user, or *both* SW and HW [42].

**Factor 4: Test Case Dependencies.** Test dependencies may lead to intermittently failing tests of embedded systems as well as for unit level testing. This was also observed by e.g., [3, 14, 43, 44] which they called a test smell. As was the case with Mandelbugs, this factor can be non-trivial for developers to identify, and several approaches to address the dependencies have been proposed, e.g. in [23, 48]. Anecdotally, test case dependencies have been seen at Westermo, but were not observed in our case study.

**Factor 5: Resource Leaks.** Resource leaks may lead to intermittently failing tests of embedded systems, and have been identified as a factor leading to flakiness in unit level testing [3, 14, 29, 43, 44]. Avritzer and Weyuker [5] found similar results for system level testing in the telecom domain. This factor has also been observed at Westermo, as mentioned in the motivational example in Section 2.2.2, but was not seen in the case study.

**Factor 6: Network Issues.** Network issues have been identified as a factor for intermittently failing tests by [3, 43]. Several of the fixes observed in the case study were related to incorrect use of link breakers in the test systems, which sometimes had a negative impact on network performance leading to intermittently failing tests. This factor is related to both test case assumptions and the complexity of testing.

**Factor 7: Random Number Issues.** Incorrect use of random numbers leading to intermittently failing tests at the system level was identified by both [3] and [14]. Some test cases at Westermo use random numbers, but it was not identified as a factor for failing tests in the current case study.

**Factor 8: Test System Issues.** Test system issues are related to the factor of complexity of testing. In our data, we saw that replacement of HW prototypes, interference in or misconfiguration of the console communication, or issues with I/O for powering HW could lead to intermittently failing tests. Alégroth and Gonzalez-Huerta, and Vahabzadeh et al. investigate technical debt and bugs in TW [4, 44]; they found that bugs and technical debt can be as present in TW as in other SW. For system level testing, technical debt and bugs in TW could very well have impacts on a test system. Wiklund et al. [46] made a literature study on impediments for software test automation and observed several challenges with respect to test systems in general, including environment configuration and quality issues (including false positives, false negatives and fragile test scripts).

**Factor 9: Code Maintenance.** In 2001, van Deursen et al. [45] spoke of problematic tests in a paper on refactoring TW. As expected, we observed that refactoring and maintenance of test code could lead to intermittently failing tests as well as to consistently failing tests.

## 6 DISCUSSION

From our case study and related work, we identified nine factors that could lead to intermittently failing tests when doing system testing of embedded systems, as well as a number of subcategories, e.g. hardware allocation and test system issues. Many of the sub-categories have been identified in previous work. The three top reasons for unit level flaky tests identified by Luo et al. [29] were asynchronous wait, concurrency and test order dependency. These overlap with the factors we identified for intermittently failing system tests. It is thus tempting to generalize findings for intermittently failing tests at the unit level to system level testing. However, we also observed important differences between their study and ours: they observed that most test cases were flaky when first written, which was not the case for us, and most of the flaky tests they investigated were flaky independent of platform whereas we saw no such indication. Instead, we observed that test cases that failed consistently seemed to do so on more than one test system.

We observed differences in root causes for test cases that fail intermittently and consistently. In particular, most tests for which the root cause was code maintenance failed consistently. In addition, test cases that failed consistently were more likely to have a shared root cause – meaning that one fix would repair several combinations of test script, parameter settings, and test system, than those that failed intermittently. We also observed that test cases that failed intermittently required more effort in terms of number of fixes than consistently failing tests. They also required more effort in terms of tools used in order to identify the root cause. However, the differences in root causes for test cases that fail intermittently and consistently are not easy to reason about. As an example, we identified poor *timing* as a factor for intermittence. It is far from trivial to argue that this, or any other root cause, would lead to either intermittent or consistent failures. Could one conclude that *all* test cases with *any* range would lead to intermittence, or only test cases with *poor* ranges, and if so, how does one identify such a range? When a test case developer designs a test case, can they easily identify a good range from a poor range? In our case study, ranges in timing and temperature led to intermittent failures, while others involving the number of lines of log file to parse led to consistent failures. Was this merely coincidental in our study, or is it because of some characteristics of those ranges? We observe that root causes leading to failing tests are not easily identified as risk factors for intermittent or consistently failing tests – only as risk factors for failing tests. We hope to explore in the future characteristics that help to identify these sorts of differences.

Lam et al. [27] investigated flaky tests in five projects and found that 0.8 to 8.4% of the test cases were flaky, whereas we found 0.98% to 3.17% of the test cases to be intermittently failing. However, q-score as a method to identify intermittently failing has a far from perfect signal to noise ratio, e.g., many of the identified test cases were still under investigation after the date range during which we

collected data. This might indicate that the thresholds of p-score and q-score should be refined, or that q-score should be used with other metrics, e.g., code churn, or for other purposes, perhaps as an indicator of the level of intermittence of a test case in the last month of testing. The number of fixed intermittently failing tests and the quality of this classification can be expected to vary with thresholds and window sizes used for p-score and q-score. Future work could investigate suitable levels further.

An interesting question of responsibility that Ahmad et al. [3] raise is whether or not a test case should include retries and thereby potentially “cover up” the flakiness, or instead expose an issue. Implementing a retry seems to be a common correction strategy [3, 9, 43]. We believe that retrying is a risky and unwise strategy that will lead to field failures because it simply masks the problem. In effect it is ignoring a known issue, even if it occurs only sporadically. Gao [19] strives for filtering away results from intermittently failing tests, so that a developer can focus on the consistently failing tests. Again, we argue that tests that retry, or tools filtering away flakiness, would hide “real issues” in SW or HW as opposed to “irrelevant” issues with TW. We therefore believe that intermittently failing tests should be considered crucial clues for investigating bugs that appear only sporadically.

Our findings imply that practitioners encountering an intermittently failing test when developing an embedded system could ask themselves:

- (1) Is the test case making incorrect assumptions on ranges or timing?
- (2) Is the actual testing context well understood?
- (3) Do test cases use shared resources without proper cleanup?
- (4) Is there a resource leak in the SW?
- (5) Are there assumptions on network performance, availability, etc., that are not always met?
- (6) Are random numbers improperly used?
- (7) Are the subsystems of the TW (test framework, the HW, or other peripheral systems used) well understood and properly configured?
- (8) Is there ongoing code maintenance or refactoring?, or
- (9) Are you observing an intermittent issue in SW or HW that needs investigation?

## 7 VALIDITY ANALYSIS

This case study relies on several constructs that originate in non-academic literature, that have not been carefully defined, or that use several overlapping definitions. Central constructs are *non-deterministic tests*, *flaky tests* and *intermittently failing tests*. We also introduced the metrics *q-score* and *p-score*. In particular, Ahmad et al. criticizes the term *flaky tests* because the shortcomings that lead to failures are not always in the tests, making the term misleading [3]. Furthermore, different studies sometimes define a flaky test with one definition, but collect data using a different metric (test cases that produce different verdicts the same week, or over 10 executions, etc.). There is thus an important threat to *construct validity*, not only in this study, but in much of the prior work on flaky and non-deterministic tests. We defined q-score as a way to measure intermittently failing tests, and collected test cases of interest using it. This has the advantage of making the study more reproducible,

but introduces the potential risk of using a metric that is not yet “proven in use.” The metric may also have improved the *reliability* of the study since other researchers with the same test results data could collect test cases of interest in the same way.

The fixes were identified by the first author using a number of tools available at Westermo, and tools developed for this study. Being one researcher in this process may be a risk to *internal validity* because identified fixes could have been different if other researchers had done the same analysis with the same data. The authors have been working at or with Westermo and company data for years, so there has been a *prolonged involvement* which would reduce the risk of poor internal validity. By using several tools in the analysis, and looking at the same data from different perspectives, there was a form of *triangulation* of the phenomena which could have reduced the threat to internal validity.

Research results that are *generalizable* can be used in other contexts, and typically industry case studies claim poor generalizability. One should thus see this study as one part of the increasing body of knowledge on intermittently failing tests in embedded systems, and not as the complete picture.

## 8 CONCLUSION

We studied intermittently failing tests in system level testing of embedded systems in a continuous integration development model. Using a novel metric, we identified groups of tests that failed intermittently and consistently. We analyzed the root causes and fixes of these tests and identified nine risk factors for intermittently failing tests: test case assumptions, complexity of testing, software or hardware faults, test case dependencies, resource leaks, network issues, random numbers issues, test system issues, and code maintenance. The most important differences between consistently failing tests and intermittently failing tests are that the intermittent tests did not always have a root cause that could be identified, intermittent tests were sometimes indicators of software or hardware faults, some intermittent tests were still under investigation after the data range we collected data from, and fixes for a consistently failing test would also often repair other tests. We also observe that many root causes of intermittence in system level testing are the same as for unit level testing.

## ACKNOWLEDGMENTS

This work was sponsored by Westermo Network Technologies AB, the Knowledge Foundation (grants 20150277, 20160139, and 20130258); the Swedish Research Council (621-2014-4925); the Swedish Innovation Agency (MegaM@Rt2); Electronic Component Systems for European Leadership (737494); and EU Horizon 2020 (grant 871319).

## REFERENCES

- [1] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. 2015. A survey on testing for cyber physical system. In *IFIP International Conference on Testing Software and Systems*. Springer.
- [2] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. 2017. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications* 8, 1 (2017), 4.
- [3] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. 2019. Empirical Analysis of Factors and their Effect on Test Flakiness—Practitioners’ Perceptions. *Preprint arXiv:1906.00673* (2019).
- [4] Emil Alégroth and Javier Gonzalez-Huerta. 2017. Towards a Mapping of Software Technical Debt onto Testware. In *Euromicro Conference on Software Engineering and Advanced Applications*. IEEE.
- [5] Alberto Avritzer and Elaine J Weyuker. 1995. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering* 21, 9 (1995).
- [6] Roorzeh Bakhti, Surya Kunche, and Michael Pecht. 2014. Intermittent failures in hardware and software. *Journal of Electronic Packaging* 136, 1 (2014), 011014.
- [7] M Ball and F Hardie. 1969. Effects and detection of intermittent failures in digital systems. In *Proceedings of the November 18–20, 1969, fall joint computer conference (AFIPS’69)*. ACM.
- [8] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2016. On Testing Embedded Software. *Advances in Computers* 101 (2016), 121–153.
- [9] Jonathan Bell, Owolabi Legunse, Michael Hilton, Lamyaa Eloussi, Tiffany Yung, and Darko Marinov. 2018. DeFlaker: automatically detecting flaky tests. In *International Conference on Software Engineering*. ACM.
- [10] Melvin A Breuer. 1973. Testing for intermittent faults in digital circuits. *IEEE Trans. Comput.* 100, 3 (1973), 241–246.
- [11] Davide G Cavezza, Roberto Pietrantuono, Javier Alonso, Stefano Russo, and Kishor S Trivedi. 2014. Reproducibility of environment-dependent software failures: An experience report. In *International Symposium on Software Reliability Engineering*. IEEE.
- [12] W Fordham Cooper. 1947. Electrical control of dangerous machinery and processes. *Journal of the Institution of Electrical Engineers-Part II: Power Engineering* 94, 39 (1947), 216–232.
- [13] Catello di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *International Conference on Dependable Systems and Networks*. IEEE/IFIP.
- [14] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer’s Perspective. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.
- [15] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering*. ACM.
- [16] Sigrid Eldh, Sasikumar Punnekkat, Hans Hansson, and Peter Jönsson. 2007. Component testing is not enough—a study of software faults in telecom middleware. In *Testing of Software and Communicating Systems*. Springer, 74–89.
- [17] Martin Fowler. 2011. Eradicating Non-Determinism in Tests (Blog Post). <https://www.martinfowler.com/articles/nonDeterminism.html>. Online, Accessed 2019-06-26.
- [18] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- [19] Zebao Gao. 2017. *Quantifying Flakiness and Minimizing its Effects on Software Testing*. Ph.D. Dissertation. University of Maryland.
- [20] Vahid Garousi, Michael Felderer, Çağrı Murat Karapıçak, and Uğur Yılmaz. 2018. What we know about testing embedded software. *IEEE Software* 35, 4 (2018), 62–69.
- [21] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* 138 (2018), 52–81.
- [22] Michael Grottkau and Kishor S Trivedi. 2005. A classification of software faults. *Journal of Reliability Engineering Association of Japan* 27, 7 (2005), 425–438.
- [23] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*. ACM.
- [24] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *International Conference on Software Engineering*, Vol. 2. IEEE.
- [25] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In *International Conference on Software Engineering*. IEEE.
- [26] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: a study of Java projects using continuous integration. In *Joint Meeting on Foundations of Software Engineering*. ACM.
- [27] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thumalapala. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *International Symposium on Software Testing and Analysis*. ACM.
- [28] Nancy G Leveson. 2004. Role of software in spacecraft accidents. *Journal of spacecraft and Rockets* 41, 4 (2004), 564–575.
- [29] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*. ACM.
- [30] Yashwant K Malaiya and Stephen YH Su. 1979. A survey of methods for intermittent fault analysis. In *International Workshop on Managing Requirements Knowledge*. IEEE.
- [31] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. 2016. Continuous integration applied to software-intensive embedded systems – problems and experiences.

- In *International Conference on Product-Focused Software Process Improvement*. Springer.
- [32] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Symposium on Operating Systems Design and Implementation*. USENIX.
  - [33] Thomas J Ostrand and Elaine J Weyuker. 1984. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software* 4, 4 (1984), 289–300.
  - [34] Nicolas Privault. 2013. *Understanding Markov chains: examples and applications*. Springer Science & Business Media.
  - [35] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. 2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.
  - [36] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943.
  - [37] Per Erik Strandberg, Wasif Afzal, Thomas Ostrand, Elaine Weyuker, and Daniel Sundmark. 2017. Automated System Level Regression Test Prioritization in a Nutshell. *IEEE Software* 34, 1 (2017), 1–10.
  - [38] Per Erik Strandberg, Wasif Afzal, and Daniel Sundmark. 2018. Decision Making and Visualizations Based on Test Results. In *International Symposium on Empirical Software Engineering and Measurement*. ACM/IEEE.
  - [39] Per Erik Strandberg, Eduard Paul Enoui, Wasif Afzal, Daniel Sundmark, and Robert Feldt. 2019. Information Flow in Software Testing – An Interview Study With Embedded Software Engineering Practitioners. *IEEE Access* 7 (2019), 46434–46453.
  - [40] Per Erik Strandberg, Thomas J Ostrand, Elaine J Weyuker, Daniel Sundmark, and Wasif Afzal. 2018. Automated test mapping and coverage for network topologies. In *International Symposium on Software Testing and Analysis*. ACM.
  - [41] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas J Ostrand, and Elaine J Weyuker. 2016. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *International Symposium on Software Reliability Engineering*. IEEE.
  - [42] Nikolaos Sycofylllos. 2016. An Empirical Exploration in the Study of Software-Related Fatal Failures. Bachelor thesis, Mälardalen University.
  - [43] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *International Conference on Software Maintenance and Evolution*. IEEE.
  - [44] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *International Conference on Software Maintenance and Evolution*. IEEE.
  - [45] Arie van Deursen, Leon Moonen, Alex van Den Berg, and Gerard Kok. 2001. Refactoring test code. In *International conference on extreme programming and flexible processes in software engineering*.
  - [46] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. 2017. Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability* (2017).
  - [47] Wayne H Wolf. 1994. Hardware-software co-design of embedded systems. *Proc. IEEE* 82, 7 (1994), 967–989.
  - [48] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*. ACM.



# Feasible and Stressful Trajectory Generation for Mobile Robots

## ABSTRACT

While executing nominal tests on mobile robots is required for their validation, such tests may overlook faults that arise under trajectories that accentuate certain aspects of the robot’s behavior. Uncovering such stressful trajectories is challenging as the input space for these systems, as they move, is extremely large, and the relation between a planned trajectory and its potential to induce stress can be subtle. To address this challenge we propose a framework that 1) integrates kinematic and dynamic physical models of the robot into the automated trajectory generation in order to generate valid trajectories, and 2) incorporates a parameterizable scoring model to efficiently generate physically valid yet stressful trajectories for a broad range of mobile robots. We evaluate our approach on four variants of a state-of-the-art quadrotor in a racing simulator. We find that, for non-trivial length trajectories, the incorporation of the kinematic and dynamic model is crucial to generate any valid trajectory, and that the approach with the best hand-crafted scoring model and with a trained scoring model can cause on average a 55.9% and 41.3% more stress than a random selection among valid trajectories. A follow-up study shows that the approach was able to induce similar stress on a deployed commercial quadrotor, with trajectories that deviated up to 6m from the intended ones.

## CCS CONCEPTS

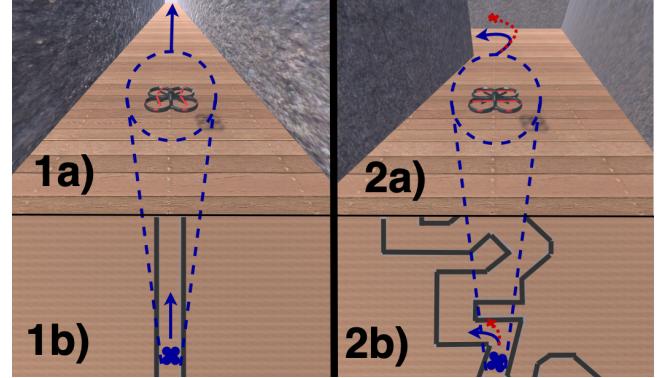
- Computer systems organization → Robotics; • Software and its engineering → Software testing and debugging; Dynamic analysis.

## KEYWORDS

Test Generation, Stress Testing, Robotics, Kinematic and Dynamic Models

## ACM Reference Format:

Carl Hildebrandt, Sebastian Elbaum, Nicola Bezzo, and Matthew B. Dwyer. 2020. Feasible and Stressful Trajectory Generation for Mobile Robots. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397387>



**Figure 1:** (1) shows a quadrotor flying in a straight corridor. (2) shows a more difficult trajectory through a winding corridor. (\*a) show the quadrotor from behind, while (\*b) show a bird’s eye view of the quadrotor. The dashed lines convey location across views, solid arrows show optimal behavior, while dotted arrows show unforeseen behavior leading to a collision.

## 1 INTRODUCTION

Mobile robots are becoming more pervasive, ranging from autonomous cars [70] to micro-inspection drones [43], and this is raising awareness of the potential impact of faults in such systems, as evident in recent crashes [5, 26]. End-to-end system testing is a standard technique to detect such faults. System tests consist of executing a trajectory that resembles future deployment environments [9, 56]. For example, with autonomous cars, trajectories can be devised over existing road maps with typical traffic loads [61], over synthetic maps that meet road-design and traffic constraints [30], or over scenarios developed following certain even probability distribution [17]. A similar procedure is utilized for testing drones while accounting for the additional third spatial dimension [56].

While exploring typical trajectories is necessary to validate the behavior of mobile robots, it may overlook faults that arise in the presence of stressful trajectories, trajectories that accentuate a particular behavior of the robot. This is analogous to the motivation for stress testing software with harsh inputs as a complementary way to judge its robustness [4]. Consider, for example, the physical maneuvers that are required for a micro-drone to race through a tight tunnel. Intuitively, a trajectory as the one on the left of Figure 1 will not subject the drone to the same level of stress as the one on the right side, which is full of tight turns that put pressure on the whole system, from the perception subsystem to the rotors.

The goal of this work is to provide an automated approach for the systematic generation of stressful trajectories for mobile robots.

Such an approach must overcome two fundamental challenges: 1) determining whether a trajectory is physically valid as a precondition to be feasible, and 2) efficiently identifying stress-inducing trajectories. A third and cross-cutting challenge is to remain general enough to support a broad range of mobile robots, stress measures, and ease of execution in both simulated and real environments.

To address the first challenge, we build on the insight that mobile robots' physical capabilities are usually approximated with kinematic and dynamic (KD) models. A KD model is a mathematical description of how the physical state of the robot is affected by any given input. For example, a KD model for a quadrotor may compute position, linear and angular velocity, and attitude based on the torque applied to the rotors. The proposed approach uses readily available KD models to compute the set of physically achievable future states of a robot[27]. Any state outside that set is physically infeasible.

To address the second challenge, we build on reachability algorithms using KD models to efficiently sample the valid physical space and incorporate a parameterizable scoring model that assigns a score representing stress to each trajectory as it is generated. Furthermore, the approach uses a beam search to incrementally explore the space of trajectories in order to identify and select the ones with the most potential to add stress to the system.

Finally, the challenge of remaining general is addressed by virtue of building on models that are available or easily approximated for most common mobile robot types, a high-degree of trajectory search abstraction and parameterization, and the use of the Robot Operating System (ROS) to standardize the message formats[59] and of open-source simulators to explore the trajectories as part of the implementation.

The primary contributions of this work are:

- 1) An automated approach for the efficient generation of physically valid and stressful trajectories for mobile robots, through the novel integration of kinematics and dynamics (system's physical aspects), the development of a parameterizable scoring model, and a configurable trajectory search process.
- 2) A tool pipeline<sup>1</sup> that implements the approach and is applicable to a broad range of mobile robots. In order to facilitate the evaluation, the implementation includes instances of an open-source quadrotor with four different software controllers and an existing commercial quadrotor.
- 3) An evaluation of the approach that demonstrates its benefits. A controlled evaluation shows that a KD model is crucial for generating trajectories of non-trivial length, while the introduction of handcrafted and learned scoring models increased stress by 56% and 41% on average over a random baseline. A case study on a commercial quadrotor demonstrated similar levels of induced stress, with the drone deviating up to 6m from its nominal trajectory.

## 2 BACKGROUND AND RELATED WORK

We begin with an overview of how mobile robots are currently being tested in §2.1, followed by an introduction to KD models in §2.2.

<sup>1</sup>Artifact available at: <https://hildebrandt-carl.github.io/RobotTestGenerationArtifact/>

### 2.1 Testing of Mobile Robots

The rise of autonomous cars and the impact of their potential failures have led to a resurgence of techniques for testing mobile robots. We point the reader to work by Stelle et al. [60] and Huang et al. [24] for overviews on techniques from the intelligent vehicle community. We now summarize efforts related to ours, organized by whether a technique is meant to operate at the system or component level, then we briefly discuss the closest work for generating stress-like tests for mobile robots, and finalize with a special mention for simulation, a key tool in the validation of mobile robots.

At the system level, the state of practice relies extensively on simulation, execution of predefined scenarios [14], and field deployment for testing [65]. For instance, in 2018, Waymo announced that it had completed 10 million miles of driving on public roads and over 7 billion miles in simulation [71]. State-of-the-art has concentrated on enabling the generation of test scenarios that account for a rich set of factors that appear in realistic contexts [57]. Many approaches have focused on the generation of images, a challenging input type that is fundamental to most mobile systems [28], and in the alteration of those images to expose faults [10, 63]. A natural progression of these efforts had led to domain-specific languages that can express images for realistic contexts[17]. Other emerging approaches target different types of inputs, such as control commands to manage acceleration, velocities, or positions [31]. All these approaches recognize that generating a full test environment is more complex and thus attempts to leverage existing information to guide test generation. For example, importing existing road maps instead of synthesizing ones [61], distilling police reports as models to guide the generation of the environment such that they resemble contexts associated with car crashes [18, 35], using an approximation to the system control model to guide the command generation [31], or incorporating traffic models [54].

Independent of the chosen approach to generate tests for mobile robots, a recognized challenge is the management of the enormous input dimensions and state-space[32]. To address these challenges existing approaches either reduce the input space, reduce the represented state space, or define a set of constraints to work within. For example, Loiacono et al. [37] use their domain knowledge to focus on race-tracks as the input space of autonomous cars. Althoff et al. [1] reduce the state space by setting the test scenario as constant and only optimizing the initial conditions of the vehicle under test. BaerkGu et al. [30] use the power of SMT solvers to generate a sequence of road segments that meet user design criteria. O'Kelly et al. [47] focused just on particular highway settings.

At the component testing level, we find many specialized input-generation techniques. For example, there are techniques targeting sensor and actuation components [8], control components [3, 40], image processing components [15, 36, 67], and reactive layers that include machine learning models. Among the latest, the software engineering community has generated an increasing body of knowledge on testing DNN's [11, 62, 77, 78].

The closest efforts to our work in terms of the integration of the system physical elements, aim to force robots to either operate along performance boundaries [45], or at maximizing exposure to unsafe behavior [1, 64]. These efforts are different from ours in two ways. First, they only aim to generate the initial set of

robot conditions instead of a whole trajectory. In theory, one could expose all valid stressful trajectories by just setting the initial robot conditions. In practice, however, identifying initial conditions that are representative of how the system operates in the real-world is challenging as they must avoid both unreachable conditions as well as conditions that are impossible to reproduce in real deployments. Second, none of them connects the KD model to generating tests that are guaranteed to be physically valid for the given robot. The work by Althoff et al. [1] makes the connection to KD models but uses them to favor tests with small reachability sets that represent tight operating spaces, which may not necessarily be stressful.

It is worth noting that simulation plays a crucial role in the validation of mobile robots driven by factors such as the time required to build a complete physical system prototype, the interactions with the physical world that require to mock at least part of that world, and the cost of field failures. Indeed, most of the approaches listed rely on various types of simulation support. Gambi et al. [19] used BeamNg[49] a vehicle simulator to find vehicle bugs using genetic algorithms. Dosovitskiy et al. [9] use a simulator, Carla, to prototype three types of autonomous vehicle. Among the many simulators available [53], in this work, we leverage recent advances in high-fidelity ones that provide not just accurate modeling of the world through sophisticated physics engines (which model, for example, gravity, friction, inertia), but also emulate a robot's sensors as it moves through the world (the atmospheric pressure, the distance to an object as measured by a laser scan, the images captured by a camera). Simulators like Carla, [9], Airsim, [56], and FlightGoggles [23] are increasingly providing such capabilities. In one of the studies in this work, we execute the generated trajectories by extending the FlightGoggles[23] framework with four additional controllers, and redesigning FlightGoggles rendering software using the Unity game engine[13] to allow the development of scenarios that do not rely on proprietary resources.

## 2.2 Kinematic and Dynamic Models

Kinematic models describe the motion of an object through measures such as position, velocity, and acceleration [2, 27, 72, 73]. Dynamic models describe the forces associated with the motion of an object [20]. Given an object's current state and a given input, these models can predict the object's future state. Such predictions are used in many fields including robotics[7], astrophysics[74], mechanical engineering[12], biomechanics[58], and game physics simulations[41]. Within the field of robotics, most systems are likely to base their development on a KD model, or can at least be approximated by an existing model.

In this work, we are specifically interested in using a KD model to describe how a robot's state  $s$  (e.g., position, velocity, acceleration) will change due to some input  $u$ . For instance a quadrotor KD model can be described using a 12<sup>th</sup> order state system  $s = [x \ y \ z \ \phi \ \theta \ \psi \ v_x \ v_y \ v_z \ \omega_x \ \omega_y \ \omega_z]^T$ , which describes the position, attitude, velocity, and angular velocity respectively [66, 75]. The input to the model is four motor speeds  $w_{1-4}$ . The speeds are used to calculate the values  $u_1$  to  $u_4$  as shown in equation 1.  $u_1$  represents the thrust force upwards  $F$  generated by the four rotors.  $u_2$  and  $u_3$  represent the difference in thrust for both roll  $M_x$  and pitch  $M_y$  respectively.  $u_4$  is the difference in torque between the two clockwise turning rotors and the two counterclockwise turning rotors which result

in yaw  $M_z$ .  $d$  is the drone's arm length, and  $k_f$  and  $k_m$  are the proportionality constants for thrust and moments respectively.

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F \\ M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & dk_f & 0 & -dk_f \\ -dk_f & 0 & dk_f & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \quad (1)$$

The values of  $u_2$ ,  $u_3$ , and  $u_4$  are used to compute the change in quadrotors angular velocity  $\omega$  using Equations 2, where the  $I$  terms correspond to the inertial properties unique to each quadrotor.

$$\begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \end{bmatrix} = \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix} + \begin{bmatrix} \frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} \quad (2)$$

The quadrotor's angular velocity  $\omega$  is then used to compute the change in the attitude of the quadrotor using Equations 3.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad (3)$$

Finally, the change in velocity is computed using Equations 4. The new velocity is used to update the position of the quadrotor.

$$\begin{bmatrix} \dot{v}_x \\ \dot{v}_y \\ \dot{v}_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} \begin{bmatrix} \cos(\phi)\cos(\psi)\sin(\theta) + \sin(\phi)\sin(\psi) \\ \cos(\phi)\sin(\theta)\sin(\psi) + \cos(\psi)\sin(\phi) \\ \sin(\theta)\sin(\phi) \end{bmatrix} u_1 \quad (4)$$

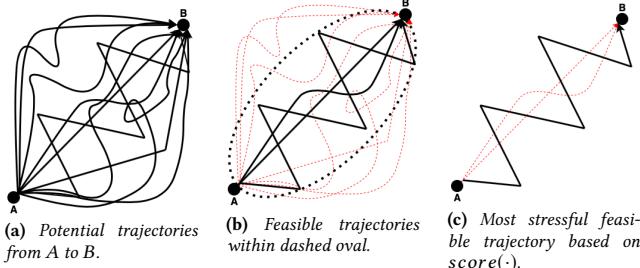
A KD model like the one introduced can be used to compute a robot reachable set, that is, the area or volume a robot can reach in a given amount of time. Efficiently calculating the reachable sets is its own research area [6, 22, 25, 34, 44, 68, 76]. For our work, a rough approximation is calculated using a sample of inputs to the KD model ( $w_{1-4}$  for the drone), and then computing the convex hull over the set of generated outputs. To the best of our knowledge, the only work that combines reachable sets with testing, minimizes the reachable sets in a given test scenario to maximize collision[1]. In contrast, we use reachable sets to create physically feasible tests.

## 3 PROBLEM STATEMENT

A physical space,  $W$ , is defined by a set of waypoints,  $wy \in W$ , with a designated origin,  $o \in W$ . A robot  $r$ , is capable of moving between a subset of waypoint pairs during a given time step,  $valid(r) \subseteq W \times W$ ; a pair  $(wy, wy') \notin valid(r)$  is said to be infeasible. A robot traversing to a waypoint,  $wy_i$ , will arrive in a state,  $s_i$ , that depends on both previous waypoint,  $wy_{i-1}$ , and previous state,  $s_{i-1}$ . For example, for a ground vehicle, a state may consist of the position  $wy$ , a velocity  $v$ , and a heading  $\theta$ .

A robot traversing through a series of waypoints is following a trajectory  $traj$ . A trajectory of length  $N_{traj}$  is a sequence of  $N_{traj}$  states,  $traj = \langle s_0, s_1, \dots, s_{N_{traj}} \rangle$ , where each state is recorded at a given  $wy$ . The  $i^{th}$   $wy$  in a trajectory is written as  $traj[i]$ .

The set of trajectories of length  $N_{traj}$ ,  $Traj$ , is exponential in size  $|Traj| = |W|^{N_{traj}}$ . Many of these trajectories are **infeasible** – they cannot be realized by the implemented system. The feasible



**Figure 2: From the set of potential trajectories, the physically feasible, and most stressful ones are selected.**

trajectories are those comprised of only valid steps,  $Traj_f = \{traj \mid \forall 0 \leq i < n : (traj[i], traj[i+1]) \in valid(r)\}$ .

Given a function,  $score : W \times W \mapsto \mathbb{R}$ , that defines the stress placed on  $r$  for a pair of waypoints according to a stress scoring criterion (e.g., maximum deviation from an intended trajectory), the stress for a trajectory is  $score(traj) = \sum_{i=0}^n score(traj[i], traj[i+1])$ . A key objective of this research is to compute the most stress-inducing feasible trajectories,  $Traj_s \in Traj_f$  such that  $\forall traj \in Traj_f : score(traj) \leq score(traj_s)$ .

This problem is illustrated in Figure 2 where Figure 2(a) depicts a set of trajectories  $Traj$ , the feasible trajectories  $Traj_f \subseteq Traj$  are shown in Figure 2(b), and given a  $score(\cdot)$  function the most stressful of the feasible trajectories,  $Traj_s \subset Traj_f$ , is found as shown in Figure 2(c).

## 4 APPROACH

The goal of the approach is the systematic and efficient generation of stressful trajectories for mobile robots. In this section we provide an overview of the approach describing the search for trajectories, a detailed description of how our approach identifies both feasible and stressful trajectories, a running example of the approach, and a brief description of the implementation.

### 4.1 Overview

The approach consists of two main algorithms. Algorithm 1 manages the search for trajectories through the use of an exploration frontier. The frontier consists of all the trajectories which are currently under consideration. Algorithm 1 expands the frontier through calls to *exploreFrontier* which is described in Algorithm 2. Algorithm 2 controls how the frontier is explored by only checking trajectories that are both feasible and stressful. In this section we take a detailed look at the workings of Algorithm 1.

Algorithm 1 manages the search for feasible stressful trajectories inside  $W$ . To keep the approach general, Algorithm 1 takes in ten parameters: (1)  $W$ : a world defining the physical volume in which trajectories will be executed, (2)  $N_{wy}$ : The number of waypoints to be explored in  $W$ , (3-4)  $wy_{start}$  and  $wy_{end}$ : a start and ending waypoints for the returned trajectories, (5)  $N_{traj}$ : the required length of a trajectory, (6)  $Limit$ : the total computation time allowed, (7)  $KD$ : the KD model of the robot, (8)  $Res$ : resolution of samples used to compute the reachable set, (9)  $Width$ : the number of trajectories explored and expanded during each loop of the algorithm, and

---

**Algorithm 1: Trajectory Generation Manager**


---

```

Input :  $W, N_{wy}, wy_{start}, wy_{end}, N_{traj}, Limit, KD, Res,$ 
         $Width, ScoringModel$ 
Output :  $Traj_s$ 
1  $Traj_s = \emptyset$ 
2 while  $time < Limit$  do
3    $Wy = randomWpSelect(W, N_{wy})$ 
4    $G_W = graph(wy_{start}, wy_{end}, Wy)$ 
5    $s_{start} = estimateRobotState(Null, Start)$ 
6    $traj_{init} = \{s_{start}\}$ 
7    $Frontier = \{(traj_{init}; 0)\}$ 
8    $Traj_c = \emptyset$ 
9   while  $Traj_c == \emptyset$  and  $|Frontier| > 0$  do
10    |  $Frontier', Traj_c = exploreFrontier(G_W, wy_{end}, KD,$ 
       |  $Frontier, Res, Width, N_{traj}, ScoringModel)$ 
11    |  $Frontier = Frontier \cup Frontier'$ 
12  end
13   $Traj_s = Traj_s \cup Traj_c$ 
14 end
15 return  $Traj_s$ 

```

---

(10) *ScoringModel*: a function used to select the most promising trajectories to be further explored.

The goal of Algorithm 1 is to find trajectories of length  $N_{traj}$ , that start and end at user-defined waypoints  $wy_{start}$  and  $wy_{end}$ . It represents  $W$  as a graph  $G_W$ , where the vertices are waypoints. Each vertex is connected to all other vertices by the shortest straight line between them, creating a complete graph. An edge represents the optimal path a robot should follow to traverse between waypoints. A path is created by combining sequences of vertices and following the edges between them. Paths through the graph represent all the possible trajectories in the world.

It starts by initializing the set of stressful trajectories  $Traj_s$  to an empty set in line 1. Algorithm 1 repeatedly computes trajectories until it exceeds a computation time of  $Limit$  and then returns the generated  $Traj_s$  in line 15. The  $Traj_s$  are computed in lines 3-13 as follows. First, in lines 2-3, a graph is built through a process used by probabilistic roadmap planners (PRM)[29]. The graph's vertices consist of  $N_{wy}$  randomly sampled waypoints as well as the user-defined  $wy_{start}$  and  $wy_{end}$  waypoints. Lines 5-7, initialize a search *Frontier*. The *Frontier* is a set of trajectories and trajectory score pairs. A trajectory score represents an estimation of the trajectory induced stress on the robot. The stress is estimated using a scoringModel described later in §4.2.3. The *Frontier* keeps track of the explored trajectories and is incrementally expanded in *exploreFrontier*. The frontier is initialized with an  $traj_{init}$  containing a single state,  $s_{start}$ , that is estimated by assuming no prior information, *Null*, and the starting waypoint  $wy_{start}$ .

The algorithm repeatedly invokes *exploreFrontier* in line 10 to incrementally expand the *Frontier* and search for complete trajectories; trajectories which start at  $wy_{start}$ , end at  $wy_{end}$ , and are length  $N_{traj}$ . Algorithm 2 describes *exploreFrontier*, which returns the newly explored frontier *Frontier'* and complete trajectories  $Traj_c$  from each iteration.

**Algorithm 2:** Explore Frontier

---

```

1 Function exploreFrontier( $G_W$ ,  $wy_{end}$ ,  $KD$ ,  $Frontier$ ,  $Res$ ,
   $Width$ ,  $N_{traj}$ ,  $ScoringModel$ )
  2    $Traj_c = \emptyset$ 
  3    $Frontier' = \emptyset$ 
  4    $SortedFrontier = \text{sort}(Frontier.\text{scores})$ 
  5   for  $i = 0$ ;  $i < Width$ ;  $i++$  do
    // Select From Frontier
    6      $traj = SortedFrontier[i].traj$ 
    7      $Frontier = Frontier \cap \text{not } traj$ 
    8     if  $|traj| == N_{traj}$ , and  $traj[N_{traj}].position == wy_{end}$ 
      then
        |  $Traj_c = Traj_c \cup traj$ 
    end
    if  $|traj| < N_{traj}$  then
      11        $last_s = traj[\text{last}].state$ 
      // Calculate Reachable Set
      13        $Reach = calculateReachSet(last_s, KD, Res)$ 
      14       for  $wy$  in ( $G_W \cap Reach$ ) do
        15          $new_s = estimateRobotState(last_s, wy)$ 
        16          $traj_n = traj \cup new_s$ 
        // Expand Frontier
        17          $Frontier' = Frontier' \cup (traj_n, \text{Null})$ 
      end
    end
    // Assign Scores
    20    $Frontier' = assignScores(Frontier', ScoringModel)$ 
  21
  end
  22    $\text{return } Frontier', Traj_c$ 

```

---

## 4.2 Efficiently Exploring the Frontier

Algorithm 2 describes the four part *exploreFrontier* function. First, *exploreFrontier* selects trajectories from the *Frontier* based on trajectory scores. Second, *exploreFrontier* computes the physical space reachable by the robot given the current robot state. Third, *exploreFrontier* expands the frontier by building a new set of trajectories by estimating the robots future state at each waypoint within the reachable space, and then using the estimated state to build new trajectories. Finally, *assignScores* gives scores to each of the new trajectories in the frontier.

More precisely, *exploreFrontier* starts by sorting the current *Frontier* based on each trajectory score. The top *Width* trajectories are selected for further processing. The larger the *Width*, the more trajectories are explored per call to *exploreFrontier*, and the more computationally expensive the operation is. However, the larger the *Width*, the more likely the algorithm will process a trajectory that will induce stress as approximated by our scoring function.

Selecting from the frontier, in line 6-10, consists of removing the  $i^{th}$  most promising trajectory and checking if it meets the requirements to be a complete trajectory. If so it is added to the *Traj<sub>c</sub>* set. In lines 11-19, if the selected trajectory is shorter than

$N_{traj}$ , the search continues by expanding the selected trajectory and adding it to *Frontier'*.

Before the selected trajectory is expanded and *Frontier'* computed, a reachable set *Reach* needs to be computed. *Reach* defines the physical space the robot can achieve in a time step given its current state. Thus all *wy* inside both  $G_W$  and *Reach* are feasible for the robot. More specifically, the reachable set is computed using the robots last known state *last<sub>s</sub>*, the robots *KD* model, and a sample resolution *Res*. Computing *Reach* is described in §4.2.1.

Once all the feasible waypoints for the robot are known, the algorithm expands the frontier. A trajectory is a sequence of states. Thus for each of the feasible waypoints, a new robot state is estimated based on the robots last known state. For each of the possible future states, a new trajectory is created by appending the new state onto the current trajectory. Each new trajectory is then added to the frontier and scores assigned to them before being returned.

**4.2.1 Reachability Analysis to Explore the Feasible Frontier.** The computed reachable set allows the algorithm to precisely identify which waypoints in  $G_W$  are achievable given the robots *KD* model and *last<sub>s</sub>*. Thus trajectories that the robot could not physically achieve can be rejected during trajectory generation, as opposed to during trajectory execution.

In this work, we explore two techniques to compute reachable sets and later compare them to a baseline technique that sets the entire space as reachable. The first approach over-estimates the reachable space, by setting the reachable set to a sphere around the current position, whose radius is equal to the maximum velocity the quadrotor can travel in  $\Delta t = 1$  s.

The second approach leverages the full *KD* model to compute the reachable set. Computing such reachable sets is an active area of research[6, 22, 25, 34, 44, 68, 76]. For simplicity, we implement a brute force technique to compute it. Given *last<sub>s</sub>*, we generate a set of input samples and apply the *KD* model to produce a set of potential reachable states. The convex hull of this state set is computed to serve as an approximation of the reachable set. This approach requires  $Res^x$  evaluations of the forward *KD* model equations, where  $x$  is the number of input variables for the *KD* model equation, and *Res* is the number of input samples taken [7]. For example, in the case of the quadrotor, which we later study, there are 4 input variables, as shown in Equation 1. If permutations of 5 linearly sampled inputs are taken, the approach would need to perform  $5^4 = 625$  computations resulting in 625 achievable future states.

**4.2.2 Estimating Robot State for Trajectory Building.** The robot's state at a new waypoint is estimated based on the robot's state at a previous waypoint and the current input. Approaches to state estimation can vary in cost and precision. At two extremes in this spectrum are estimators that 1) assume the robot is *at rest* when reaching a waypoint, and 2) solve the inverse of the *KD* model equations. The first is inexpensive, but imprecise and the second is precise, but expensive.

We implement a hybrid of these that uses only portions of the *KD* model equations to estimate state while setting the remaining state variables to their resting values. The portions of the state to reset are configurable. For example, for the drone systems we later study, the approach computes quadrotor velocity at each waypoint

**Algorithm 3:** Assign Scores

---

```

1 Function assignScores(Frontier, ScoringModel)
2   for traj in Frontier do
3     score = 0
4     for each pairOfStates in traj do
5       score += scoringModel(pairOfStates)
6     end
7     traj.score = score
8   end
9   return Frontier

```

---

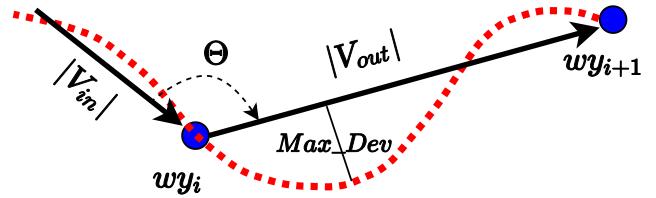
(euclidean distance between the waypoints over the timestep), but sets the attitude and angular velocity to 0 at each waypoint (This would happen if a quadrotor entered a waypoint level).

**4.2.3 Assigning Scores to Select Next Trajectory.** The *scoringModel* is used as described in Algorithm 3. For each *traj* in the *Frontier* we start with an initial score of 0. The algorithm iterates through each pair of states in the trajectory and assigns a score to the state pair. The final trajectory score is then computed by accumulating the state pair scores for that trajectory.

The scores are assigned by a *scoringModel* and are calculated based on an estimate of the stress that the robot will incur. A good *scoringModel* will accurately estimate this stress given two robot states and associated waypoints. The stress is defined through a scalar stress metric. Depending on the application of the robot, stress can be measured using different stress metrics. For example, three possible metrics are maximum deviation, maximum acceleration, or total time. The only requirement is that the selected stress metric must be measurable during robot execution. The main stress metric we use in our study is maximum deviation, which is illustrated in Figure 3. The maximum deviation, a standard measure associated with navigation safety, is a measure of the largest error between the expected position of a robot and its actual position. In this work, we explored two classes of scoring models that we later compare to a baseline scoring model that randomly selects a score.

The first scoring model leverages a user's domain knowledge to create rules likely to maximize some goal, for instance in our case, maximizing deviation. In our evaluation, for example, we identified the trajectories velocity  $v_{in}$ ,  $v_{out}$ , and the trajectory angle  $\Theta$ , as shown in Figure 3, as attributes likely to be correlated to maximum deviation. For example, a large  $v_{in}$  and  $\Theta$  correspond with the intuition that entering a waypoint with high velocity might result in a significant deviation if the robot is also required to take a sharp turn. In general, the effectiveness of such a model will depend on a domain expert's ability to identify the attributes as well as how closely the attributes align with the robot behavior, which depends on the robot planner, robot controller, and robot sensing and actuation capabilities.

The second scoring model learns from previous data. It consists of using a collection of trajectories generated using a random scoring model and subsequently identifying the factors that lead to particularly stressful trajectories. This knowledge can then be used to score future trajectories on their ability to cause stress. As an example, assume that there is a series of generated trajectories. The



**Figure 3: Trajectory attributes and the stress metric maximum deviation.** The solid line is the expected trajectory while the dotted line is the true behavior.

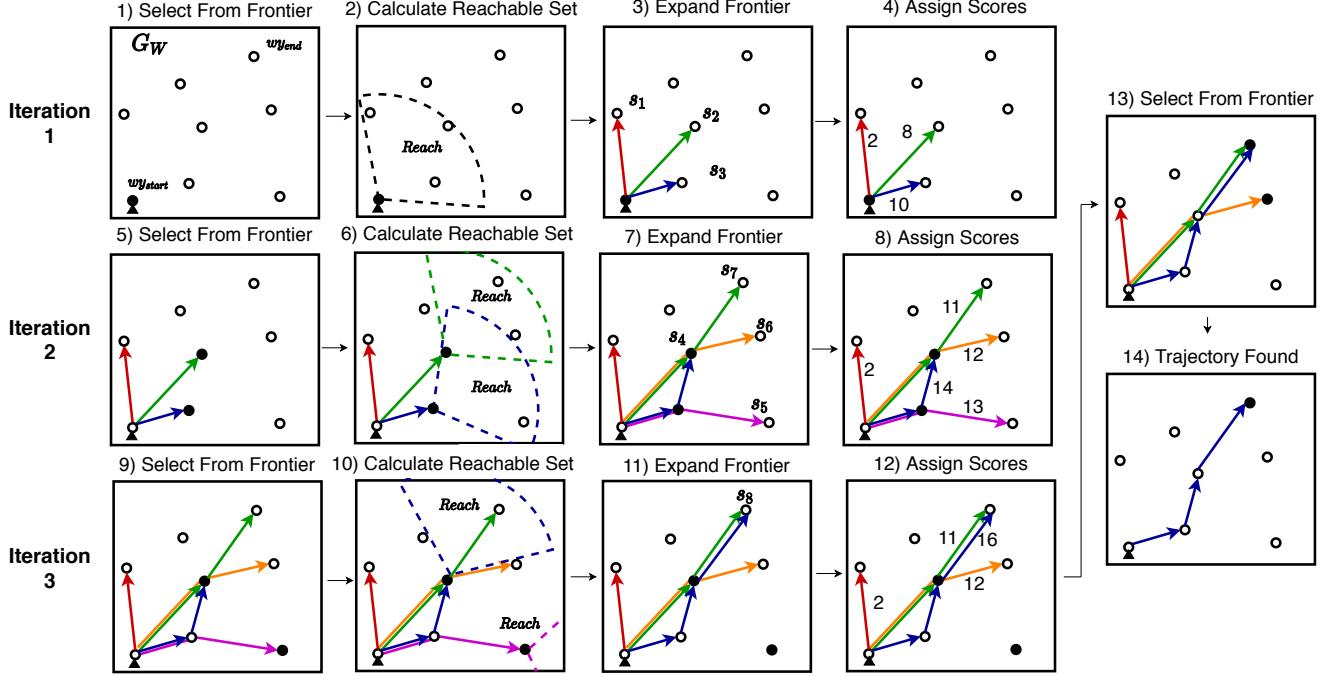
robot could then execute the trajectories to render an actual maximum deviation. The traversed trajectories could then be broken down into pairs of waypoints like that of Figure 3. The maximum deviation associated with each pair of waypoints *max\_dev* and a set of attributes that may be associated with that deviation (e.g.,  $v_{in}$ ,  $v_{out}$ ,  $\Theta$ ) could be used as training data. Then a learning technique can be used to produce a *scoringModel* that, given a pair of waypoint attributes, can estimate the expected maximum deviation.

In our evaluation, we generated a *scoringModel* using a polynomial regression model where the loss function is the linear least-squares function, and regularization is given by the  $\ell_2$ -norm[21]. We determined the best polynomial degree using 10-fold cross-validation. If the resulting model provides a good fit (i.e., strong correlation and low cross-validation loss), then it can be used to assign predictive scores to future trajectories without executing them. This approach incurs the cost of trajectory execution to generate the data to train the model. Thus its applicability depends in part on the cost of such execution. In many cases, such costs can be mitigated, for example through simulation, and it is beneficial in that it does not rely on the user's expertise.

### 4.3 Example Trajectory Generation

Figure 4 shows a step-by-step illustration of our approach. In this example, the  $G_W$  is generated using 6 random waypoints, we set *Width* to 2, and  $N_{\text{traj}}$  to 4. After PRM construction, we select from the frontier, which after the initialization in Algorithm 1 lines 3–8, is a single trajectory that contains  $wy_{\text{start}}$ . We calculated the *Reach* for the last and only waypoint ( $wy_{\text{start}}$ ) in the trajectory as described in §4.2.1. We then expanded the frontier using each of the waypoints inside the  $W$  and *Reach*. The waypoints are added to the current trajectory by estimating three new states based on the current state and the new waypoint as described in §4.2.2. Scores are assigned to each of the new trajectories based on a scoring model as described in §4.2.3.

On the second iteration, due to the *Width* of 2, the two highest-scoring trajectories are selected from the frontier (filled circle). For each of the selected trajectories last waypoints, a *Reach* is calculated. The frontier is expanded using the waypoints in each *Reach*. This results in 4 new trajectories. Note that a waypoint can be used in multiple trajectories, as seen by the most central waypoint, which



**Figure 4:** Our approach illustrated with waypoints (circles), trajectories (solid lines), and reachable sets (dotted lines). The example considers a 2D world with 6 random waypoints, a beam width of 2, and a trajectory length of 4. The mobile robot in this example, starts with 0 velocity and is facing directly upward (small triangle).

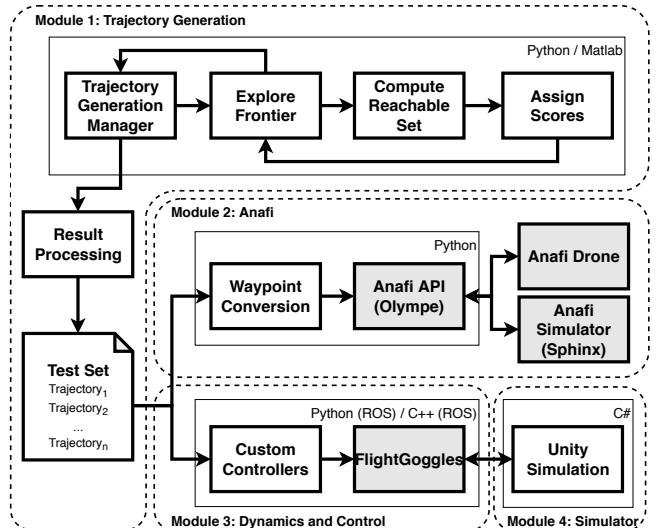
at this point is part of three trajectories. Scores are assigned to each trajectory and trajectories selected from the frontier.

On the third iteration the most central waypoint is again processed as it is the last state in the trajectory with a score of 14. Note however, the resultant *Reach* is different as the trajectory selected arrived at the waypoint with a different final state. The second *Reach* is computed for the trajectory with a score of 13. There are no waypoints inside this *Reach*, and thus the trajectory is removed from the frontier. Scores are assigned to the new trajectories and trajectories selected from the frontier. Two trajectories with a score of 16 and 12 are chosen for processing. The trajectory with score 16 meets the criteria of starting at  $w_{start}$ , ending at  $w_{end}$ , and being of length 4. This trajectory is added to the  $Traj_c$ , and the algorithm repeated with a new  $G_W$ . Although this is a hypothetical example, the approach still selects a stressful trajectory. The final trajectory requests the robot to take an immediate  $\approx 70$  degree right turn, followed by a  $\approx 45$  degree left turn before moving to  $w_{end}$ .

#### 4.4 Implementation

The implementation consists of 4 main software modules<sup>2</sup>, as seen in Figure 5. The first module, trajectory generation, implements the approach as described in §4.1, while the next three modules run the experiments and are vital for collecting the data used later in the study.

<sup>2</sup>Artifact available at: <https://hildebrandt-carl.github.io/RobotTestGenerationArtifact/>



**Figure 5:** An overview of the implementation. Existing software is highlighted in a darker shade.

The first module consisted of both the trajectory generation and result processing toolchain. The trajectory generation uses the trajectory manager to explore the frontier using the reachable set and scoring model. The resultant trajectories are then processed and converted into data files and images that can be accessed by

both the Anafi and FlightGoggles control software. The majority of this module is implemented using Python3. The module consists of 36 python scripts with a total of approximately 7,000 SLOC. For certain functions, such as computing the convex hull, it was more convenient to use MATLAB, and so the approach calls these functions through the MATLAB API for Python[39].

The second module implements the integration with the Anafi quadrotor in both simulation[51] and the real-world through the Anafi API[50]. The module consists of software used to convert the trajectory into waypoints that are readable by the Anafi API. The Anafi API sends the waypoints to the Anafi quadrotor and records the returned GPS data through either a virtual ethernet or Wi-Fi connection.

The final two modules contain the control and simulator code to fly the FlightGoggles quadrotor. The FlightGoggles simulator has two parts. The first part emulates the dynamics and control of the quadrotor, while the second part simulates the quadrotors sensor data and collision information. At the time of writing, the FlightGoggles simulation uses proprietary graphics assets. Thus we only use the part of FlightGoggles that emulates the quadrotor dynamics, and we re-engineer the FlightGoggles simulation tool in Unity based on the available documentation. The control code uses ROS[59] and is written in C++. The implementation of the 4 custom quadrotors is integrated into the original code base using 11 Python classes consisting of approximately 2150 SLOC. The portions of the FlightGoggles simulator that were redeveloped in Unity are written in C#. The new simulator integrates with the base ROS code using the original TCP link in FlightGoggles. The new simulator uses assets that are freely available from the Unity store.

## 5 EVALUATION

The goal of the evaluation is to assess the proposed approach and determine what benefits the introduction of the KD model and scoring models has on automated trajectory generation for robots. More specifically, we aim to answer the following research questions for automated trajectory generation:

**RQ1)** Does the introduction of a KD model improve the ability to generate feasible and valid trajectories?

**RQ2)** Does the introduction of a scoring model improve the ability to generate stressful trajectories?

### 5.1 Setup

The test world is set to a  $30m \times 30m \times 30m$  map with 250 randomly placed waypoints. This selection matches the volume ( $27000m^3$ ) and size of a typical outdoor aerial testing facility[33, 46, 69].

**5.1.1 Robot Configurations.** The systems we used are listed in Table 1. The first is an autonomous racing quadrotor executed in the publicly available FlightGoggles simulator [23]. The quadrotor has a weight of  $1kg$ , and a body length of  $0.45m$  [55]. Its maximum velocity in simulation is  $18m/s$  [38].

The FlightGoggles quadrotor comes with a built-in angular rate controller to manage roll, pitch, and yaw. To evaluate the wide variety of trajectory following techniques exhibited by today’s quadrotors, we implement four commonly used quadrotor controllers[66] into the FlightGoggles simulator. Two controllers are of a waypoint control type, using a cascade of three PID controllers; the

**Table 1: Robot configurations studied**

Robot Hardware	Robot Software	Execution
Flightgoggles Quadrotor[23]	Unstable Waypoint Controller[66]	Simulation
	Stable Waypoint Controller[66]	Simulation
	Fixed Velocity Controller	Simulation
	Minimum Snap Controller[42]	Simulation
Parrot Anafi Quadrotor [48]	Waypoint Controller[50]	Simulation
		Real World

first controls the angle of the quadrotor, the second controls the velocity of the quadrotor using the angle controller, the third sets the velocity of the quadrotor based on the distance to a waypoint. The first implementation replicated poorly written controllers that has overshoot and oscillation around waypoints. The second implementation mimics tuned controllers that are stable and converge to the waypoint. The next instantiated controller was a fixed velocity controller. This controller assigns a shared proportion of a fixed velocity over each the x, y, and z-direction based on the location of the next waypoint. We set the controller to maintain a velocity of  $2m/s$ , allowing the quadrotor to maneuver easily. The final controller computed a minimum snap trajectory and follows it using the waypoint PID controller. It was fundamentally different in that it builds a new trajectory through the waypoints that minimize snap, the 4th derivative of position[42], which means that it does not adhere to the assumption of the expected behavior being the shortest straight line between consecutive waypoints.

A second quadrotor, the Anafi Parrot, is studied later in §6. Trajectories on the Anafi Parrot were executed both in simulation and the real-world using its proprietary control software.

### 5.2 Trajectory Generation with KD Models

To answer RQ1, we need to assess the cost and benefit of incorporating a KD model into the trajectory generation technique. To the best of our knowledge, there are currently no automated approaches or tools available for the automated generation of stressful target trajectories for mobile robots. The state-of-the-practice consists of handcrafted stress tests built by experts, which tend to be effective but limited in the scale of exploration. Thus, to identify the benefits explicitly introduced by the KD models, we adapted how the reachable set in line 13 of Algorithm 2 is computed using 3 different techniques. The first approach, **No KD**, returns all waypoints in the world, without considering any form of a KD model. The second approach weakly approximates the reachable set, **Approx KD**, by computing a sphere whose radius is the distance the quadrotor could travel at maximum velocity in  $\Delta t = 1s$ . The final approach, **Full KD**, uses a full KD model as described in §4.2.1. While expensive [22], this guarantees that all explored trajectories are valid by construction.

Each technique was given 2 hours to generate and execute trajectories. Algorithm 2 was set to have a beamwidth of 5 and trajectory length between 3 and 50. Varying trajectory length allows us to

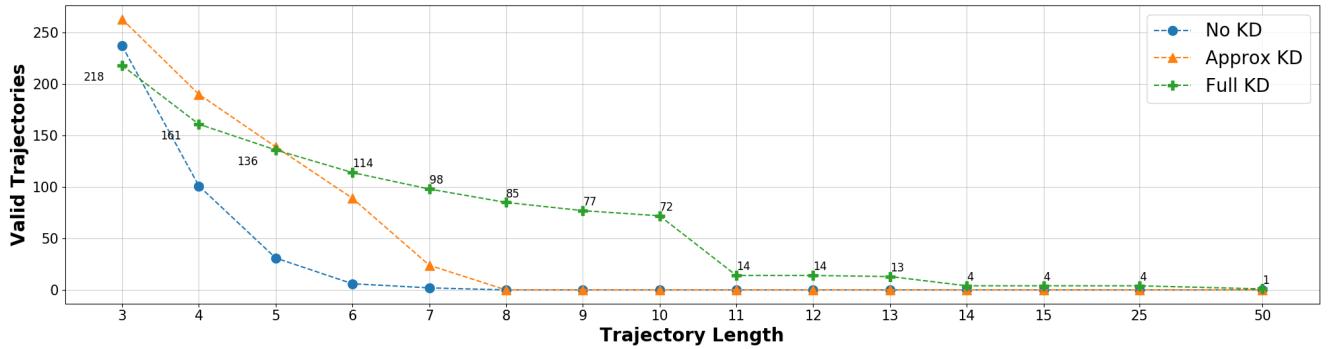


Figure 6: Valid trajectories generated of varying length.

assess the efficiency of techniques as the problem scales in complexity. For example, the number of possible trajectories of length 3 in a world with 250 possible waypoints is  $1.5 \times 10^7$  and that increases to  $4.1 \times 10^{17}$  for trajectories of length 50.

For each technique, trajectories returned in line 22 of Algorithm 2 were checked for validity using the robot full KD model. For each valid trajectory, we model its execution time in proportion to its length and decrement the total experiment time. The number of physically valid trajectories returned by each technique within the 2 hour limit is shown in Figure 6.

Figure 6 shows that for simpler trajectories of length 3 for **No KD** and trajectories of length 5 for **Approx KD** the computationally cheaper approach produces more valid trajectories as opposed to our **Full KD** approach. This is because generating short, physically valid trajectories is easier, as only a few valid waypoints need to be selected. However, we can see that as the trajectories become longer and more complex, it becomes beneficial to use the computationally more expensive **Full KD** model. Figure 6 shows that **Full KD** start to outperform both **No KD** and **Approx KD** for trajectories of length 4 and 6 respectively, in terms of the number of physically valid trajectories produced. In fact, for trajectories of length 8 both **No KD** and **Approx KD** are unable to produce any valid trajectories in the given amount of time, while the **Full KD** can produce 85 valid trajectories. Even for trajectories of length 50, **Full KD** can still find 1 valid trajectory in the given time. This is because the **Full KD** approach provides information to Algorithm 2 on which waypoints in the world lead to invalid trajectories. This information, although expensive to generate, allows the search technique to reject invalid trajectories during trajectory generation as opposed to trajectory execution. This is especially important since the number of possible trajectories grows exponentially with their length – making pruning invalid paths cost-effective.

We then computed several performance metrics for valid trajectories of length 10. We ran each of the valid trajectories from the **Full KD** approach using the FlightGoggles simulator with the stable waypoint controller. Figure 7 shows the distribution of 3 performance metrics, namely: maximum deviation ( $m$ ) from the optimal trajectory, the maximum acceleration ( $m/s^2$ ) of the robot, and total execution time ( $s$ ). We chose these because they are diverse in that deviation captures the potential for the robot to operate unsafely,

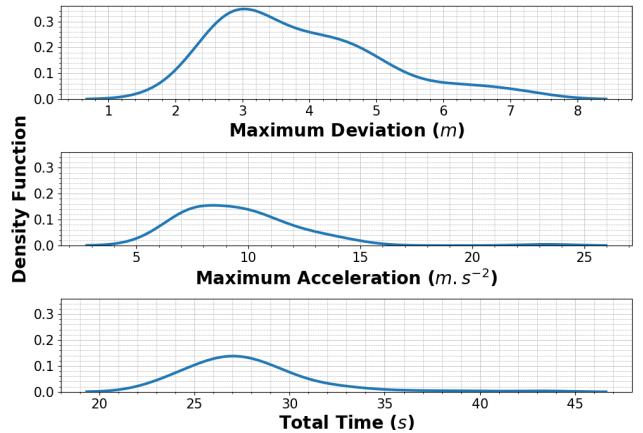


Figure 7: The distribution of performance metrics obtained by executing the FlightGoggles quadrotor in simulation.

acceleration captures the stress placed on the robot hardware, and total time reflects the robots ability to operate effectively.

Figure 7 shows that for each of the metrics, the quadrotor exhibits a broad range of possible values and that each of the distributions is positively skewed, with long tails to the right (where there is more stress). The fact that the distribution has long tails to the right shows that even though most trajectories produce little stress, there are trajectories which significantly stress the robot and lie outside the normal operating profile. Figure 7 shows that not only do the valid trajectories with no scoring model result in a range of behavior but that we are also able to measure multiple performance metrics on them.

**RQ1 Findings:** Although it is computationally more expensive to use a KD model, incorporating it into trajectory generation is critical for efficiently generating **valid** trajectories, especially as the trajectory length increases. We also found that, independent of the chosen measure, the stress induced valid generated trajectories shows high variability.

**Table 2: The different scoring models and their descriptions.**

<b>High Velocity</b>	Assigns high scores to trajectories with high velocities.
<b>High Velocity + 90 Deg</b>	Assigns high scores to trajectories with high velocities and include 90 degree turns.
<b>High Velocity + 180 Deg</b>	Assigns high scores to trajectories with high velocities and include 180 degree turns
<b>Learned</b>	Learns a scoring model based on the execution of prior trajectories

### 5.3 Incorporating a Scoring Model

To answer RQ2, we need to determine whether computing and including a scoring model, line 20 of Algorithm 2, leads to the generation of more stressful trajectories. We explore 4 different scoring models as described in Table 2. The first 3 scoring models are designed to represent scoring models designed by experts. Intuition tells us that for a quadrotor, the higher a robot’s velocity, the more deviation we can expect given a turn. Using this intuition, three handcrafted scoring metrics were created. The first assigned higher scores to **High Velocity** trajectories without consideration to turns. The second assigned higher scores to trajectories that had both high velocity and waypoints that resulted in 90 degree turns (**High Velocity + 90 Deg**). The last handcrafted scoring model was similar to the second, except it placed a high score on 180 degree turns (**High Velocity + 180 Deg**).

These three approaches require domain knowledge, which is not always readily available. We thus tried a final scoring model, which **Learned** a scoring model based on the maximum deviation of each controller on the initial trajectories in RQ1. The learned scoring model uses 10-fold cross-validation to determine the polynomial degree used in a ridge regression model implemented using Python’s Scikit-Learn library[52]. For each of the software controllers tested in RQ2, we extract attributes from their initial execution. The input and output velocity, the angle between the waypoints, and the actual maximum deviation is extracted, as shown in Figure 3. Using this as training data, we produced four independent scoring models that, given a pair of waypoints, predict the maximum deviation for the respective software controller.

For each new scoring model, we generated a new set of trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length 10. That is half of the time given in the RQ1 study to determine if the scoring model could produce more stressful resultant trajectories and do so in less time. For comparison, we also generated a baseline where each of the FlightGoggles software controllers was executed on the trajectory set generated using a **Full KD** model and no scoring model as per RQ1 with trajectories of length 10 and 2 hours of generation.

The resulting trajectories were run on each of the drone controllers, and the maximum deviation recorded.<sup>3</sup> To determine whether the introduction of a scoring model was beneficial, we divided each of the resultant maximum deviations with the mean maximum deviation from the baseline trajectory set. Thus any test that induced more stress and had a maximum deviation greater than the initial

<sup>3</sup>Due to space constraints and without loss of generality, we just use the maximum deviation since it relates to safety – the further a quadrotor is away from the expected trajectory, the more significant the safety risk.

test set with no scoring model from RQ1, would result in a value greater than 1. Similarly, a test with a value of less than 1 means that it induced less stress than the average test in RQ1.

The results are shown in Figure 8. When considering only the handcrafted scoring models, Figure 8 shows that for each of the controllers, at least 1 of the 3 handcrafted scoring models results in a more stressful test set. For both waypoint controllers, including a scoring model that favors trajectories of high velocity result in test sets that are 70% and 76% more stressful. For the fixed velocity controller, a scoring model that favors 180 degree turns resulted in a test set that is 10% more stressful. The low increase in stress is attributed to the controller’s slow constant speed, however, we note that our approach still finds test cases that are ≈40% more stressful than the given random test set. For the minimum snap controller a scoring model that favored 90-degree turns induces on average 69% more stress. These findings are consistent with the operation of these controllers. Moreover, taking the mean of the best scoring models shows that, on average, having a handcrafted scoring model results in a **55.9% increase in maximum deviation** on the stressful trajectories. These findings show that handcrafted scoring models are beneficial when domain knowledge is available.

Figure 8 also shows that for all controllers, it is possible to learn scoring models that can generate stressful trajectories for a specific quadrotor. This is useful, especially when there is no domain knowledge available, for instance, when testing a new robot. Moreover, the quality of learned models is high, since for each controller we found the learned model produced a distribution of performance metrics similar to the best handcrafted scoring model. Taking the mean of all scoring models showed that on average a learned scoring model **increased the maximum deviation by 41.3%**.

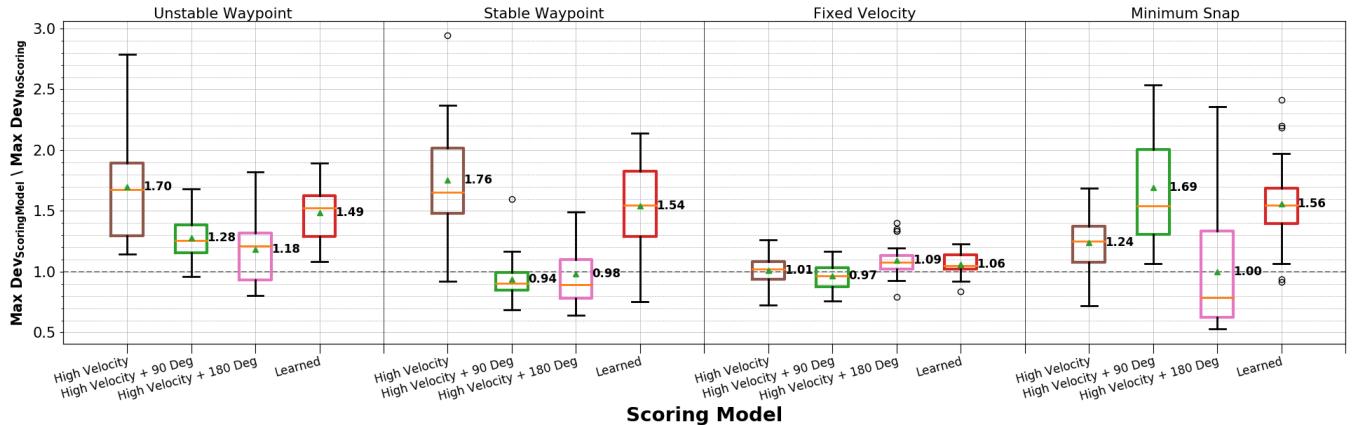
Recall that the experimental setup for RQ2 used half of the time compared to RQ1, so the observed improvements in the performance metrics were also significantly less costly to produce.

**RQ2 Findings:** Introducing both handcrafted and learned scoring model into trajectory generation produces test that on average are **55.9% and 41.3% more stressful** than trajectories without a scoring model respectively. Moreover, learned scoring models can be generated without any prior domain knowledge.

## 6 FOLLOW-UP STUDY

We performed a preliminary study to explore the application of the proposed approach to a commercial drone operating in an outdoor flying cage of  $30m \times 30m \times 30m$ , and analyzed the differences between executing the trajectories in simulation versus the real-world. We selected the popular Parrot’s Anafi quadrotor[48], which has a weight of  $0.5kg$ , maximum horizontal velocity of  $15m/s$ , and an arm length of  $0.1m$ . The Anafi has an autonomous flight mode, which can follow a series of waypoints through change positions commands using a controller that is not publicly available.

For this study, since we are not certain about the particular controller used by the Anafi, we learned a scoring model from an initial set of trajectories that we executed sending waypoints to Anafi’s API. To reduce the cost of collecting the training set of trajectories,



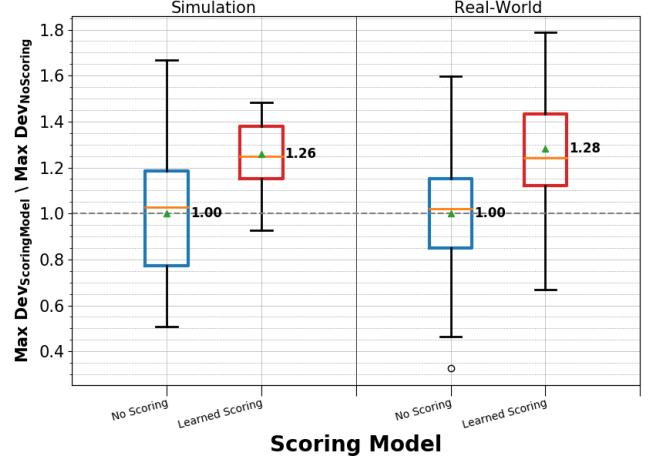
**Figure 8:** The ratio of maximum deviation with a scoring model to maximum deviation without one. Here the initial trajectory set with no scoring model would have a mean value of 1. Any trajectory set that produced more stress than the initial trajectory set would have values greater than 1. The medians (central line) and mean (triangle and number) are shown.

we executed those initial set of random trajectories in the Parrot-Sphinx[51] simulator. We bound the initial generation to 2 hours, a beamwidth of 5, and a trajectory length of 10, and the learning was meant to generate a model that increases the maximum deviation in a trajectory. We then used the learned scoring model to generate stress-inducing trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length of 10.

Figure 9 shows the findings in the form of a boxplot. The first pair of boxes show the results from the execution of trajectories in simulation, while the second pair of boxes show the results from executing the drone in the real world. Each pair represents the deviation of the initial trajectory set and the stress-inducing trajectory set, respectively. Each box is normalized by the mean maximum deviation of the corresponding initial trajectory set. As mentioned earlier, the initial trajectory set was generated without a scoring model, and it shows similar means and variation in simulation and in the real world.

As shown by the second box, the scoring model learned in simulation allows our approach to generate trajectories that, when executed in simulation, cause on average a 26% increase maximum deviation in a trajectory. More interesting, however, is that when the same generated trajectories are executed in the real-world, they also cause a similar degree of additional deviations, albeit with greater variation (whiskers of the fourth box) introduced by external environmental factors such as GPS-localization noise and wind. This confirms that it is possible to mitigate the cost of learning a scoring model through simulation and apply trajectories generated with that model in real-world contexts.

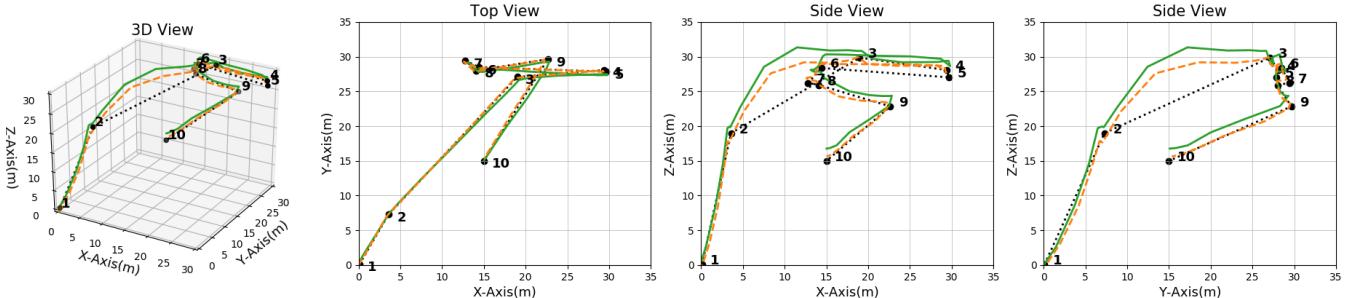
From a testing point of view, one might be interested in trajectories (test inputs) that violated certain specifications. For example, might specify that the maximum deviation from the expected trajectory cannot exceed some threshold. Figure 11 shows the percentage of automatically generated tests that violate a given maximum distance specification. The results indicate that, regardless of the specified maximum deviation, using a scoring model produces a larger percentage of tests that violate the specification. For example, given a specified maximum deviation of  $4m$ , Figure 11 shows



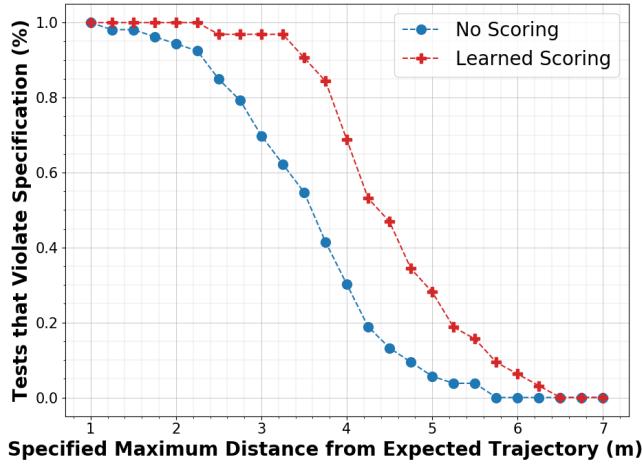
**Figure 9:** Maximum deviation for simulation and outdoors trajectories normalized by the mean of the trajectory set with no scoring model.

that with no scoring model, only 30% of tests generated would violate that constraint. However, our approach using a scoring model would generate a test set with approximately 70% of tests that violate the same specification. Additionally, these results show that using a scoring model not only generates a higher percentage of tests that violate the constraints, but also generates the test with the largest maximum deviation. The maximum deviation we observed outdoors with the generated trajectories was  $6.2m$ , with the average being  $4.5m$ . This highlights how the approach can generate stressful trajectories that push the drone to deviations that go way beyond the expected deviation for this kind of drone.

Developers can also use these trajectories to further investigate the behaviors which led to these violations. For example, using the Anafi quadrotor, we plotted the test that produced the largest maximum deviation. Figure 10 shows the generated test trajectory



**Figure 10:** Anafi’s position in the real-world and simulation as it traverses one of the stress-inducing trajectories. The expected behaviour is marked as dots. The simulated data is marked with dashes. The real-world data is a solid line.



**Figure 11:** The percentage of outdoor tests which violated the specified maximum deviation

(dotted line) of the drone in both simulation (dashed line) and real-world (solid line). From the top view, it appears that the Anafi follows the expected trajectory precisely. However, from a side view, it seems like the Anafi follows the expected trajectory in all cases except when large changes in all x,y, and z-directions are requested, for instance, when flying from waypoint 2 to 3. The 3rd waypoint is the position (19.0, 27.0, 29.9). In simulation, although it did not follow the expected short line trajectory, it flew to a height of 29.9m as expected. In the real world, the Anafi similarly did not follow the expected trajectory, however it flew to a height of  $\approx$ 31.34m high, 1.34m over the designated flying altitude of 30m, even though all waypoints are within the flying volume. A pilot flying this quadrotor who was not aware of the distinct behavior shown through this trajectory would at best be surprised and, at worst, experience a collision.

## 7 CONCLUSION

We have introduced a novel approach for the automatic generation of feasible and stressful trajectories for mobile robots. The approach is unique in that it integrates the kinematics and dynamics of the robot to generate trajectories that are feasible given its physical limitations, it builds on algorithms from robotics planning and

graph exploration to more efficiently search the input space, and it incorporates a highly parameterizable scoring model to guide trajectory generation towards those that induce high-stress in the system. The approach was able to generate valid trajectories that caused a mean increase of maximum deviations of 55.9% and 41.3% in the two systems we studied. These deviations are significant as in the commercial quadrotor, the maximum deviation recorded in a small  $30m^3$  area was 6m.

In future work, we will investigate reduction techniques that let us explore the trajectory space more efficiently by, for example, exploiting the physical commutativity of sub-trajectories. Another exciting avenue that we intend to explore is combining our trajectory generation technique with languages used to specify environments[16]. Using this, we hypothesize that we could generate not only trajectories but entire environments that stress any given robot. These environments could be combined with a mix-mode execution where, for example, the drone flies in the real world, but the obstacles are present only in simulation.

## ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Awards #CCF-1853374 and #CCF-1901769 as well as the U.S. Army Research Office Grant #W911NF-19-1-0054. We thank the developers of FlightGoggles[23] for making their systems and software available.

## REFERENCES

- [1] Matthias Althoff and Sebastian Lutz. 2018. Automatic Generation of Safety-Critical Test Scenarios for Collision Avoidance of Road Vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1326–1333.
- [2] Joseph Stiles Beggs. 1983. *Kinematics*. CRC Press.
- [3] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 63–74.
- [4] Robert Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional. 745–746 pages.
- [5] Brian Garrett-Glaser. 2019. Avionics - Drone Delivery Crash in Switzerland Raises Safety Concerns As UPS Forms Subsidiary. <https://www.aviationtoday.com/2019/08/08/drone-delivery-crash-in-switzerland-raises-safety-concerns/>. [Online; accessed 5-November-2019].
- [6] Mo Chen, Sylvia Herbert, and Claire J Tomlin. 2016. Fast reachable set approximations via state decoupling disturbances. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 191–196.
- [7] Gregory S Chirikjian. 1996. Synthesis of Discretely Actuated Manipulator Workspaces via Harmonic Analysis. In *Recent Advances in Robot Kinematics*. Springer, 169–178.

- [8] Anders Lyhne Christensen, Rehan O'Grady, Mauro Birattari, and Marco Dorigo. 2008. Fault detection in autonomous robots based on fault injection and learning. *Autonomous Robots* 24, 1 (2008), 49–67.
- [9] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938* (2017).
- [10] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2017. Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. *arXiv:1703.00978 [cs.SY]*
- [11] Tommaso Dreossi, Shromona Ghosh, Alberto Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2017. Systematic Testing of Convolutional Neural Networks for Autonomous Driving. *arXiv:1708.03309 [cs.CV]*
- [12] Homer D Eckhardt. 1998. *Kinematic design of machines and mechanisms*. McGraw-Hill New York.
- [13] Unity Game Engine. 2008. Unity game engine-official site. *Online* [Cited: October 9, 2008.] <http://unity3d.com> (2008), 1534–4320.
- [14] Michelle Chaka Eric Thorn, Shawn Kimmel. 2018. A Framework for Automated Driving System Testable Cases and Scenarios. [https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13882-automateddrivingsystems\\_092618\\_v1a\\_tag.pdf](https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13882-automateddrivingsystems_092618_v1a_tag.pdf).
- [15] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. 2017. *Learning to recognize distance to stop signs using the virtual world of Grand Theft Auto 5*. Technical Report.
- [16] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–78.
- [17] Daniel J Fremont, Xiangyu Yue, Tommaso Dreossi, Shromona Ghosh, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2018. Scenic: Language-based scene generation. *arXiv preprint arXiv:1809.09310* (2018).
- [18] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 257–267.
- [19] Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 318–328.
- [20] Maxime Gautier. 1986. Identification of robots dynamics. *IFAC Proceedings Volumes* 19, 14 (1986), 125–130.
- [21] Aurélien Géron. 2017. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.".
- [22] Antoine Girard and Colas Le Guernic. 2008. Efficient reachability analysis for linear systems using support functions. *IFAC Proceedings Volumes* 41, 2 (2008), 8966–8971.
- [23] Winter Guerra, Ezra Tal, Varun Murali, Gilhyun Ryou, and Sertac Karaman. 2019. FlightGoggles: Photorealistic Sensor Simulation for Perception-driven Robotics using Photogrammetry and Virtual Reality. *arXiv preprint arXiv:1905.11377* (2019).
- [24] WuLing Huang, Kunfeng Wang, Yisheng Lv, and FengHua Zhu. 2016. Autonomous vehicles testing methods review. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 163–168.
- [25] Inseok Hwang, Dušan M Stipanović, and Claire J Tomlin. 2005. Polytopic approximations of reachable sets applied to linear dynamic games and a class of nonlinear systems. In *Advances in control, communication networks, and transportation systems*. Springer, 3–19.
- [26] Jackie Wattles. 2019. CNN Business - Tesla on Autopilot crashed when the driver's hands were not detected on the wheel. <https://www.cnn.com/2019/05/16/cars/tesla-autopilot-crash/index.html>. [Online; accessed 5-November-2019].
- [27] Reza N Jazar. 2010. *Theory of applied robotics: kinematics, dynamics, and control*. Springer Science & Business Media.
- [28] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. 2016. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983* (2016).
- [29] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation* 12, 4 (1996), 566–580.
- [30] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi, and Masahiro Yamaura. 2016. The SMT-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.
- [31] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFUZZER: finding input validation bugs in robotic vehicles through control-guided testing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 425–442.
- [32] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.
- [33] Kurt Barnhart. 2015. Partners Kansas State University Salina and Westar Energy build one of the largest enclosed flight facilities for UAS in the nation. <https://www.k-state.edu/media/newsreleases/oct15/pavilion101415.html>. [Online; accessed 22-August-2019].
- [34] Alex A Kurzhanskiy and Pravin Varaiya. 2006. Ellipsoidal toolbox (ET). In *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE, 1498–1503.
- [35] Xuehua Liao, Zhousen Zhu, Yusong Yan, and Tao Lv. 2012. Traffic accident reconstruction technology research and simulation realization. In *2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM)*. IEEE, 152–155.
- [36] J. Liebelt and C. Schmid. 2010. Multi-view object class detection with a 3D geometric model. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1688–1695.
- [37] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. 2011. Automatic track generation for high-end racing games using evolutionary computation. *IEEE Transactions on computational intelligence and AI in games* 3, 3 (2011), 245–259.
- [38] Massachusetts Institute of Technology. 2019. Flight Goggles. <https://flightgoggles.mit.edu>. [Online; accessed 01-January-2020].
- [39] Mathworks. 2020. Matlab and Python. <https://www.mathworks.com/products/matlab/matlab-and-python.html>. [Online; accessed 26-January-2020].
- [40] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. 2013. Automated model-in-the-loop testing of continuous controllers using search. In *International Symposium on Search Based Software Engineering*. Springer, 141–157.
- [41] Mitch McCaffrey. 2017. *Unreal Engine VR Cookbook: Developing Virtual Reality with UE4*. Addison-Wesley Professional. Chater 7: Character Inverse Kinematics.
- [42] Daniel Mellinger and Vijay Kumar. 2011. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2520–2525.
- [43] microdrones. 2019. Microdrones Inspection Service. <https://www.microdrones.com/>. [Online; accessed 5-November-2019].
- [44] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. 2005. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on automatic control* 50, 7 (2005), 947–957.
- [45] Galen E Mullins, Paul G Stankiewicz, and Satyandra K Gupta. 2017. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1443–1450.
- [46] Nicole Casal Moore. 2019. M-Air autonomous aerial vehicle outdoor lab opens. <https://news.umich.edu/m-air-autonomous-aerial-vehicle-outdoor-lab-opens/>. [Online; accessed 22-August-2019].
- [47] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, John Duchi, and Russ Tedrake. 2018. Scalable End-to-End Autonomous Vehicle Testing via Rare-event Simulation. *arXiv:1811.00145 [cs.LG]*
- [48] Parrot. 2019. Anafi. <https://www.parrot.com/us/drones/anafi>. [Online; accessed 11-November-2019].
- [49] Parrot. 2019. Bebop 2. <https://beamng.gmbh/research/>. [Online; accessed 26-January-2020].
- [50] Parrot. 2019. Olympe Documentation. <https://developer.parrot.com/docs/olympe/>. [Online; accessed 20-November-2019].
- [51] Parrot. 2019. Parrot-Sphinx. <https://developer.parrot.com/docs/sphinx/whatissphinx.html>. [Online; accessed 22-August-2019].
- [52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [53] Francisca Rosique, Pedro Navarro Lorente, Carlos Fernandez, and Antonio Padilla. 2019. A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research. *Sensors* 19 (02 2019), 648. <https://doi.org/10.3390/s19030648>
- [54] Aritsha Sarkar and Krzysztof Czarnecki. 2019. A behavior driven approach for sampling rare event situations for autonomous vehicles. *CorR abs/1903.01539* (2019). *arXiv:1903.01539 http://arxiv.org/abs/1903.01539*
- [55] Thomas Sayre-McCord, Winter Guerra, Amado Antonini, Jasper Arneberg, Austin Brown, Guilherme Cavalheiro, Yajun Fang, Alex Gorodetsky, Dave McCoy, Sebastian Quilter, et al. 2018. Visual-inertial navigation algorithm development using photorealistic camera simulation in the loop. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2566–2573.
- [56] Shital Shah, Debadatta Dey, Chris Lovett, and Ashish Kapoor. 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*. Springer, 621–635.
- [57] Thierry Sotiropoulos, Jérémie Guiochet, Félix Ingrand, and Hélène Waeselynck. 2016. Virtual worlds for testing robot navigation: a study on the difficulty level. In *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 153–160.
- [58] Dimitar Stanev and Konstantinos Moustakas. 2019. Modeling musculoskeletal kinematic and dynamic redundancy using null space projection. *PloS one* 14, 1 (2019).
- [59] Stanford Artificial Intelligence Laboratory et al. [n.d.]. *Robotic Operating System*. <https://www.ros.org>

- [60] Jan Erik Stellet, Marc René Zofka, Jan Schumacher, Thomas Schamm, Frank Niewels, and J Marius Zöllner. 2015. Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 1455–1462.
- [61] TASS International. 2019. PreScan - A Simulation and Verification Environment for Intelligent Vehicle Systems. <https://tass.plm.automation.siemens.com/prescan>. [Online; accessed 22-August-2019].
- [62] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.
- [63] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. 2018. Simulation-based Adversarial Test Generation for Autonomous Vehicles with Machine Learning Components. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. 1555–1562.
- [64] Cumhur Erkan Tuncali, Theodore P Pavlic, and Georgios Fainekos. 2016. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1470–1475.
- [65] University of Michigan. 2020. Mcity. <https://mcity.umich.edu>. [Online; accessed 26-January-2020].
- [66] Kimon P Valavanis and George J Vachtsevanos. 2015. *Handbook of unmanned aerial vehicles*. Springer.
- [67] David Vazquez, Antonio M Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. 2013. Virtual and real world adaptation for pedestrian detection. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 797–809.
- [68] Abraham P Vinod, Bairavran HomChaudhuri, and Meeko MK Oishi. 2017. Forward stochastic reachability analysis for uncontrolled linear systems using fourier transforms. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. ACM, 35–44.
- [69] Virginia Tech. 2018. Virginia Tech Drone Park officially open . <https://vtnews.vt.edu/articles/2018/04/ictas-droneparkopens.html>. [Online; accessed 22-August-2019].
- [70] Waymo LLC. 2019. Waymo - Self Driving Car. <https://waymo.com>. [Online; accessed 5-November-2019].
- [71] Waymo Team. 2018. Where the next 10 million miles will take us. <https://medium.com/waymo/where-the-next-10-million-miles-will-take-us-de51bebb67d3>. [Online; accessed 1-December-2019].
- [72] Edmund Taylor Whittaker. 1988. *A treatise on the analytical dynamics of particles and rigid bodies*. Cambridge university press.
- [73] Thomas Wallace Wright. 1898. *Elements of mechanics including kinematics, kinetics and statics, with applications*. D. Van Nostrand Company.
- [74] Yaroslav S. Yatskiv. 2007. Kinematics and Physics of Celestial Bodies. <https://www.springer.com/journal/11963>. [ISSN: 0884-5913].
- [75] Esen Yel, Tony X Lin, and Nicola Bezzo. 2017. Reachability-based self-triggered scheduling and replanning of uav operations. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 221–228.
- [76] Esen Yel, Tony X Lin, and Nicola Bezzo. 2018. Self-triggered adaptive planning and scheduling of uav operations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 7518–7524.
- [77] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.
- [78] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic testing of driverless cars. *Commun. ACM* 62 (03 2019), 61–67. <https://doi.org/10.1145/3241979>



# Detecting Cache-Related Bugs in Spark Applications

Hui Li\*

Dong Wang\*

State Key Lab of Computer Sciences, Institute of Software,  
Chinese Academy of Sciences  
University of Chinese Academy of Sciences  
Beijing, China  
[{lihui2012,wangdong18}@otcaix.iscas.ac.cn](mailto:{lihui2012,wangdong18}@otcaix.iscas.ac.cn)

Yu Gao

Wensheng Dou†

Lijie Xu

State Key Lab of Computer Sciences, Institute of Software,  
Chinese Academy of Sciences  
University of Chinese Academy of Sciences  
Beijing, China  
[{gaoyu15,wsdou,xulijie}@otcaix.iscas.ac.cn](mailto:{gaoyu15,wsdou,xulijie}@otcaix.iscas.ac.cn)

Tianze Huang\*

Beijing University of Posts and Telecommunications  
Beijing, China  
[huangtianze@bupt.edu.cn](mailto:huangtianze@bupt.edu.cn)

Wei Wang

Jun Wei

Hua Zhong

State Key Lab of Computer Sciences, Institute of Software,  
Chinese Academy of Sciences  
University of Chinese Academy of Sciences  
Beijing, China  
[{wangwei,wj,zhonghua}@otcaix.iscas.ac.cn](mailto:{wangwei,wj,zhonghua}@otcaix.iscas.ac.cn)

## ABSTRACT

Apache Spark has been widely used to build big data applications. Spark utilizes the abstraction of Resilient Distributed Dataset (RDD) to store and retrieve large-scale data. To reduce duplicate computation of an RDD, Spark can cache the RDD in memory and then reuse it later, thus improving performance. Spark relies on application developers to enforce caching decisions by using *persist()* and *unpersist()* APIs, e.g., which RDD is persisted and when the RDD is persisted / unpersisted. Incorrect RDD caching decisions can cause duplicate computations, or waste precious memory resource, thus introducing serious performance degradation in Spark applications. In this paper, we propose *CacheCheck*, to automatically detect cache-related bugs in Spark applications. We summarize six cache-related bug patterns in Spark applications, and then dynamically detect cache-related bugs by analyzing the execution traces of Spark applications. We evaluate CacheCheck on six real-world Spark applications. The experimental result shows that CacheCheck detects 72 previously unknown cache-related bugs, and 28 of them have been fixed by developers.

## CCS CONCEPTS

- Computer systems organization → Cloud computing; Reliability;
- Software and its engineering → Software testing and debugging; Software maintenance tools.

\*Hui Li, Dong Wang and Tianze Huang are equal contributors to the paper.

†Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397353>

## KEYWORDS

Spark, cache, bug detection, performance

## ACM Reference Format:

Hui Li, Dong Wang, Tianze Huang, Yu Gao, Wensheng Dou, Lijie Xu, Wei Wang, Jun Wei, and Hua Zhong. 2020. Detecting Cache-Related Bugs in Spark Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397353>

## 1 INTRODUCTION

Apache Spark [50, 51] has become one of the most popular big data processing systems, and has been widely used in many big Internet companies, e.g., Facebook [18], Alibaba [2], eBay [19]. As an in-memory big data framework, Apache Spark can provide order of magnitude of performance speedup over disk-based big data frameworks, e.g., Apache Hadoop [1]. Especially, Apache Spark works better on iterative analytics, e.g., machine learning [37], and graph computation [33], which requires data to be shared across different iterations.

A Spark application consists of two kinds of operations: *transformation* (e.g., *map* and *filter*) and *action* (e.g., *count* and *take*). Spark utilizes the abstract of *Resilient Distributed Dataset* (RDD) to store and retrieve large-scale data, which is a read-only collection of objects partitioned across a set of machines. Transformations are used to create RDDs, and actions are used to obtain actual computation results on RDDs. Once an action completes, all its used RDDs are discarded from memory. To reuse an RDD loaded by an action, Spark provides *persist()* API to manually cache<sup>1</sup> the loaded RDD. Thus, the following actions can reuse the cached RDD, and avoid duplicate computations. Note that, an RDD can be persisted in different cache levels, e.g., in memory and on disk. Spark provides *unpersist()* API to manually release cached RDDs when they

<sup>1</sup>We use cache and persist interchangeably in this paper.

will not be used by following actions, thus saving precious memory. Figure 1a shows an example about how to reuse RDD *words* (Section 2.1).

Apache Spark relies on developers to enforce caching decisions through *persist()* and *unpersist()* APIs. Developers need to clearly figure out *which* RDD should be cached, *when* the RDD is persisted and unpersisted. The lazy evaluation of RDDs in Spark (Section 2.3) further complicates cache mechanisms, e.g., a *persist()* operation must be performed before its corresponding *action* operation. Improper caching decisions on RDDs usually cause performance degradation, even worse, out of memory errors. For example, if an RDD used by two actions is not persisted, the RDD will be computed twice, and thus degrading the application performance (e.g., SPARK-1266 [3]). If a cached RDD is not used anymore, it occupies much precious memory, and affects the normal execution (e.g., SPARK-18608 [16]). In our experiment, improper caching decisions can seriously degrade the performance of Spark applications (0.1% – 51.6%), and even cause Out of Memory (OOM). In this paper, we refer to improper caching decisions on RDDs as *cache-related bugs*.

To improve the performance of Spark applications, existing studies mainly focus on improving memory usage in the Spark system, e.g., Yak [39], AstroSpark [45] and Simba [31], and optimizing cache mechanisms in the Spark system, e.g., Neutrino [47], LRC [49] and Lotus [43]. However, cache-related bug patterns and detection approaches in Spark applications have not been studied. Note that, Spark applications have the different execution model from traditional programs (e.g., C/C++ and Java). Thus, the memory / resource leak detection techniques in traditional programs [32, 34, 40] cannot be directly applied on cache-related bug detection in Spark applications.

In this paper, we first analyze the execution semantics of cache mechanisms in Spark, and summarize six bug patterns for cache-related bugs in Spark applications, i.e., *missing persist*, *lagging persist*, *unnecessary persist*, *missing unpersist*, *premature unpersist*, and *lagging unpersist* (Section 3). We then propose *CacheCheck* to automatically detect cache-related bugs in Spark applications. After running a Spark application (without any modification or instrumentation to the application) on our modified Spark system (Section 5), *CacheCheck* collects the application’s execution trace and caching decisions. Then, *CacheCheck* infers the correct caching decisions for the application based on the execution semantics of cache mechanisms in Spark. Finally, by analyzing the difference between the original caching decisions and inferred correct caching decisions, *CacheCheck* reports the inconsistencies as cache-related bugs.

We implement *CacheCheck*, and make it publicly available at <https://github.com/Icysandwich/cachecheck>. We evaluate its effectiveness on real-world Spark applications, including GraphX [22] and MLlib [23] and Spark SQL [29], etc. Experimental results show that: (1) *CacheCheck* can precisely detect all 18 known cache-related bugs in Spark applications. (2) *CacheCheck* detects 72 previously-unknown cache-related bugs on the latest versions of six Spark applications. We have reported these cache-related bugs to the concerned developers. So far, 53 have been confirmed by developers, of which 28 have been fixed. Developers also express their great interests in *CacheCheck*.

We summarize the main contributions of this paper as follows.

- We summarize six cache-related bug patterns in Spark applications, which can seriously degrade the performance of Spark applications.
- We propose an automated approach, *CacheCheck*, to detect cache-related bugs in Spark applications.
- We implement *CacheCheck* and evaluate it on real-world Spark applications. The experimental results show that it can detect cache-related bugs effectively.

The remainder of this paper is organized as follows. Section 2 explains Spark programming model, cache mechanism, and Spark execution semantics briefly. Section 3 presents the six cache-related bug patterns in Spark applications. Section 4 presents our cache-related bug detection approach. Section 5 presents our *CacheCheck* implementation, and Section 6 evaluates *CacheCheck* on real-world Spark applications. Section 7 and Section 8 discuss threats and related work, and Section 9 concludes this paper.

## 2 BACKGROUND

In this section, we briefly introduce the programming model and the cache mechanism in Apache Spark.

### 2.1 Spark Programming Model

Spark uses the abstract of Resilient Distributed Dataset (RDD) to store and retrieve data in a distributed cluster. An RDD is an immutable data structure that is divided into multiple partitions, and each of them can be stored in memory or on disk across machines. Developers can perform two types of operations on RDDs: *transformation* (e.g., *map*) and *action* (e.g., *count*). Transformations create new RDDs from the existing ones with the specific computation (e.g., user-defined function). Actions return a value (not RDD) by performing a necessary computation (e.g., *count*) on an existing RDD.

A Spark application usually creates RDDs from input data, executes several transformations on RDDs, and performs actions to obtain computation results when necessary [35]. Figure 1a shows a code snippet that contains two jobs: obtaining the total number of words in a text file (Line 4) and taking the first 10 words in the file (Line 5). In this example, we create a new RDD *data* by loading a text file from HDFS (Line 1). Then, we split *data* into *words* by using transformation *flatMap* (Line 2). To get the total number of words, we perform the action *count* on *words* (Line 4). To obtain the first 10 words, we perform the action *take* on *words* (Line 5).

**Lineage graph:** In Spark, all transformations are lazy. When we perform transformations on RDDs, Spark does not immediately perform the transformation computation. Instead, Spark logs all computation dependencies between RDDs in a DAG (Directed Acyclic Graph), which is referred to as lineage graph. Spark keeps on building this lineage graph until an action operation is applied on the last RDD. After that, Spark generates a job to execute each transformation in the lineage graph.

For example, in Figure 1a, *sc.textFile()* (Line 1) and *data.flatMap()* (Line 2) are not executed by Spark immediately. They will only get executed once we perform an action on RDD *words*, i.e., *words.count()* (Line 4). All the transformations in Figure 1a are included in a lineage graph as shown in Figure 1b. The action operation triggers

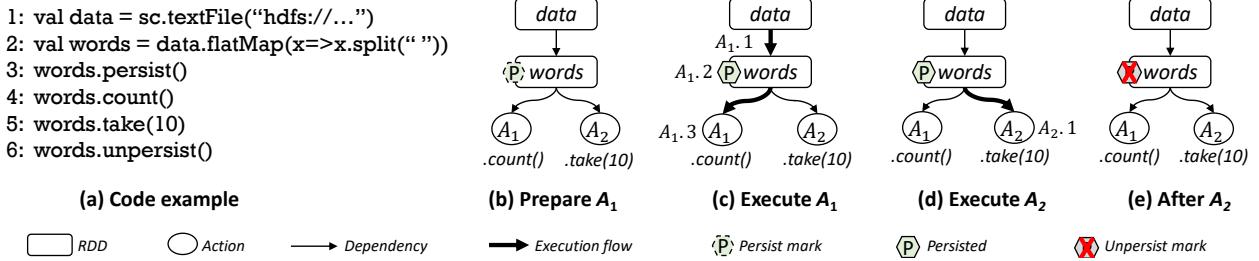


Figure 1: The execution process and cache mechanism in Spark.

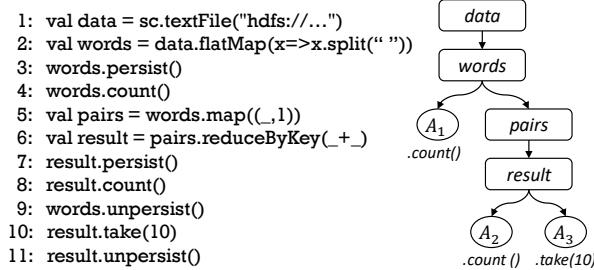


Figure 2: Word count example. The left side shows its code, and the right side shows its lineage graph.

the application execution. In other words, the created DAG will be committed to the cluster and the application begins to execute.

## 2.2 Cache Mechanism in Spark

In the process of an action execution, the transformations that the action depends on are actually computed and the transformed RDDs are materialized. Once the execution completes, these interim RDDs will be released by the Spark system. Therefore, each transformed RDD should be recomputed each time when an action runs on it. As shown in Figure 1a, RDD *data* and *words* are used twice by two actions: *words.count()* (Line 4) and *words.take(10)* (Line 5). Since each action generates a job, these two RDDs will be computed twice, thus increasing the execution time.

To speed up applications that use certain RDDs multiple times, Spark allows developers to cache data for reuse. By using cache-related APIs, these interim RDDs can be kept in memory (by default) or on disk. So, whenever we invoke another action on the cached RDD, no recomputation takes place. There are two APIs for caching an RDD: *cache()* and *persist(level : StorageLevel)*. The difference between them is that *cache()* persists the RDD into memory, whereas *persist(level)* can persist the RDD in memory or on disk according to the caching strategy specified by *level*. For easy presentation, we consistently use *persist()* to represent these two APIs in this paper. Freeing up space from the memory is performed by API *unpersist()*. In Figure 1a, *words* is cached by calling *persist()* on it (Line 3). After the execution of *words.count()* (Line 4), the data of RDD *words* is stored in memory. When executing *words.take(10)* (Line 5), we do not need to recompute *data* and *words* anymore. When *words* is no longer needed, we remove the cached RDD from memory immediately by invoking *unpersist()* on it (Line 6).

## 2.3 Spark Execution Semantics

As discussed earlier, when we call a transformation, only the lineage graph is built. When we perform a cache-related API on an reusable RDD, Spark only marks that the RDD should be persisted. The actual caching process takes place when the first action is executed on the RDD. When the next action is executed, Spark finds that the RDD has been cached and thus does not need to recompute it. If this reusable RDD is not cached, recomputation on this RDD takes place and all the RDDs it depends on may be recomputed, too.

Figure 1b shows the dependency relationships between RDDs used by action  $A_1$  (Line 4 in Figure 1a) and action  $A_2$  (Line 5 in Figure 1a). In Figure 1b, *words* is marked as persisted (Line 3 in Figure 1a). When  $A_1$  is triggered, it tries to find all involved RDDs in the lineage graph from bottom to top. Then the actual execution is performed at the sequence of  $A_{1.1}$ ,  $A_{1.2}$  and  $A_{1.3}$  in Figure 1c. Since *words* is marked as persisted, after *words* is computed, *words* is stored in the memory ( $A_{1.2}$ ). When  $A_2$  is triggered, it tries to find all involved RDDs first. Since *words* is already persisted, there is no need to find involved RDDs that *words* depends on. The actual execution of action  $A_2$  starts from *words* ( $A_{2.1}$  in Figure 1d). After the execution of action  $A_2$ , *words* is freed by Line 6 in Figure 1a, as shown in Figure 1e.

## 3 CACHE-RELATED BUG PATTERNS

In Spark, which RDD should be persisted and when the RDD is persisted / unpersisted are totally decided by developers. However, developers may misunderstand their Spark applications and the caching mechanisms, and introduce cache-related bugs. To combat cache-related bugs, we thoroughly investigate the Spark programming model and cache mechanism. Further, we enumerate all possible ways to utilize cache-related APIs, and evaluate their performance impacts. All cases that can slowdown Spark applications are considered as cache-related bug patterns. Finally, we summarize six cache-related bug patterns.

In this section, we use the word count example shown in Figure 2 to illustrate cache-related bug patterns. RDD *data* is created by loading a text file from HDFS (Line 1), and further split into *words* (Line 2) by white spaces. We obtain the number of words by performing action *count* on *words* (Line 4). Transformation *map* maps each word in *words* with 1 to create a new key-value RDD *pairs* (Line 5), and transformation *reduceByKey* further shuffles *pairs* based on the key and obtains RDD *result* that contains the number for each particular word (Line 6). We further obtain the total number of unique words in the file by performing action *count* on *result* (Line 8), and

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: +words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9: +words.unpersist()
10: result.take(10)
11: result.unpersist()

```

(a) Missing persist

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: +words.persist()
4: words.count()
5: - words.persist()
6: val pairs = words.map((_,1))
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10: words.unpersist()
11: result.take(10)
12: result.unpersist()

```

(b) Lagging persist

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: - pairs.persist()
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10: words.unpersist()
11: result.take(10)
12: - pairs.unpersist()
13: result.unpersist()

```

(c) Unnecessary persist

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9: +words.unpersist()
10: result.take(10)
11: result.unpersist()

```

(d) Missing unpersist

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: - words.unpersist()
7: val result = pairs.reduceByKey(_+_)
8: result.persist()
9: result.count()
10: +words.unpersist()
11: result.take(10)
12: result.unpersist()

```

(e) Premature unpersist

```

1: val data = sc.textFile("hdfs://...")
2: val words = data.flatMap(x=>x.split(" "))
3: words.persist()
4: words.count()
5: val pairs = words.map((_,1))
6: val result = pairs.reduceByKey(_+_)
7: result.persist()
8: result.count()
9: +words.unpersist()
10: result.take(10)
11: result.unpersist()
12: - words.unpersist()

```

(f) Lagging unpersist

**Figure 3: Cache-related bug examples and corresponding fix methods for the word count example in Figure 2.**

obtain the first 10 records in *result* by performing action *take(10)* on *result* (Line 10). Note that, *words* and *result* are persisted (Line 3 and Line 7). They are correct decisions for us. Then, we use six bug examples in Figure 3 to illustrate why other persist decisions are incorrect and how they can degrade performance.

### 3.1 Missing Persist

A transformed RDD in an action will be released by default when the action completes. If an RDD is used by the following actions, it will be recomputed multiple times, thus degrading the execution performance. To improve the execution performance, developers should persist these RDDs that can be used multiple times.

Figure 3a introduces a missing persist bug. In Figure 3a, RDD *data* and *words* are used by two actions *words.count()* (Line 4) and *result.count()* (Line 8). If *words* is not persisted, *data* and *words* will be computed twice. We do not need to persist *data* if *words* is persisted, because if *words* is persisted, *data* will not be recomputed. Therefore, the best caching decision is to persist *words* before the first action (i.e., *words.count()*) that depends on *words* is triggered. This bug slows down the example by 21.4%. For another real-world example, the missing persist bug SPARK-16697 [13] in MLlib [23] slows down its application by 51.6% (More details about performance slowdown can be found in Table 1).

### 3.2 Lagging Persist

As discussed earlier, an RDD is only marked as persisted when the *persist()* API is invoked. The actual caching process takes place when the RDD is materialized by an action. Therefore, for an RDD

that needs to be persisted, the *persist()* API must be called on the RDD before the first action that depends on it is triggered. Otherwise, the RDD cannot be persisted as expected.

Figure 3b introduces a lagging persist bug. The *persist()* operation on RDD *words* (Line 5) is invoked after action *words.count()* and before action *result.count()* (Line 9). When action *words.count()* at Line 4 is executed, *words* is not marked as persisted, and will not be persisted. When action *result.count()* is executed, *data* and *words* are computed again, since *words* is not cached yet. In this example, lagging persist of *words* causes recomputation of *data* and *words* in action *result.count()*. This bug slows down the example by 22.9%.

### 3.3 Unnecessary Persist

Cached RDDs usually occupy large precious memory. Therefore, if an RDD is not reused by multiple actions, it should not be persisted. Unnecessary persist can occupy extra memory, and affects the application execution, thus causing performance degradation.

Figure 3c introduces an unnecessary persist bug on RDD *pairs*, which is persisted at Line 6. Since RDD *result* has been persisted after the execution of action *result.count()* (Line 9). The following action *result.take()* will use cached *result* rather than *pairs*. Thus, it is unnecessary to persist *pairs* in this example. This bug slows down the example by 4.5%. The real-world unnecessary persist bug SPARK-18608 [16] slows down its application by 23.3%.

### 3.4 Missing Unpersist

If an RDD is persisted but not unpersisted when it will not be used anymore, the RDD will be kept in memory until the application

completes or it may be replaced by other cached RDDs. Thus, the unpersisted RDD can occupy precious memory, and affects the application execution, thus causing performance degradation.

Figure 3d introduces a missing unpersist bug. RDD *words* stays in memory until the application completes. In fact, after action *result.count()* at Line 8, *words* will not be used anymore. Thus, it should be unpersisted at Line 9. This bug slows down the example by 0.9%. The real-world missing unpersist bug SPARK-3918 [6] in MLlib [23] causes an OOM in its application.

### 3.5 Premature Unpersist

A cached RDD should be unpersisted until it will not be used anymore. Premature unpersist of a cached RDD will invalidate the cached RDD, and cause its recomputation, thus slowing down the application.

Figure 3e introduces a premature unpersist bug. RDD *words* is unpersisted after action *words.count()* and before action *result.count()* at Line 6. Thus, action *result.count()* at Line 9 has to recompute *words*. Only after executing action *result.count()*, RDD *words* will not be used anymore and can be unpersisted (Line 10). This bug slows down the example by 35.7%.

### 3.6 Lagging Unpersist

Once a cached RDD will not be used anymore, it should be unpersisted right away. Otherwise, the cached RDD still occupies precious memory, and affects the application execution, thus causing performance degradation.

Figure 3 introduces a lagging unpersist bug. After executing action *result.count()*, RDD *result* is persisted and RDD *words* will not be used anymore. So, RDD *words* should be unpersisted right after action *result.count()* (Line 9). However, RDD *words* is kept in memory until it is unpersisted at Line 12. This bug slows down the example by 0.3%.

## 4 DETECTING CACHE-RELATED BUGS

Given a Spark application, CacheCheck works in three steps. First, after running the application, CacheCheck collects its actual execution trace about its lineage graph, actions and caching decisions (Section 4.1). Second, from collected lineage graph and actions, CacheCheck infers the correct caching decisions based on the execution semantics of Spark (Section 4.2). Third, CacheCheck analyzes the actual and correct caching decisions, and identifies the difference, which usually indicates cache-related bugs (Section 4.3).

### 4.1 Execution Trace

The execution of a Spark application consists of a number of actions. In order to detect cache-related bugs, we need to record which RDDs are persisted, and when they are persisted and unpersisted. Thus, we only consider three kinds of operations in the execution trace of a Spark application. They are listed as follows.

- *action*: We use *rdd.action()* to denote that an *action* is performed on *rdd*.
- *persist*: We use *rdd.persist()* to denote that a *persist* operation is performed on *rdd*.
- *unpersist*: We use *rdd.unpersist()* to denote that an *unpersist* operation is performed on *rdd*.

An execution trace of a Spark execution can be represented as follows.

$$\tau = (\text{action}|\text{persist}|\text{unpersist})^*$$

In the execution trace, each action contains its partial lineage graph that it directly depends on. For example, in Figure 2, action *A*<sub>1</sub> contains its partial lineage graph *G*<sub>1</sub> = *data* → *words* → *A*<sub>1</sub>, and action *A*<sub>2</sub> contains its partial lineage graph *G*<sub>2</sub> = *data* → *words* → *pairs* → *result* → *A*<sub>2</sub>, and action *A*<sub>3</sub> contains its partial lineage graph *G*<sub>3</sub> = *data* → *words* → *pairs* → *result* → *A*<sub>3</sub>. Note that, Spark generates a unique ID for each RDD. In the execution trace, the recorded lineage graph is built on these unique IDs. For easy presentation and understanding, we use variable names instead of these unique IDs in the lineage graphs.

We can further combine the partial lineage graphs of all actions to generate the whole lineage graph *G*. The whole lineage graph generation algorithm is straightforward. For each action *A*<sub>*i*</sub>, we inspect each edge *e* in its lineage graph *G*<sub>*i*</sub>. If *e* does not exist in *G*, we merge edge *e* and its corresponding nodes into *G*. For example, we can combine *G*<sub>1</sub>, *G*<sub>2</sub>, and *G*<sub>3</sub>, and generate the lineage graph in the right side of Figure 2.

Note that, the execution trace precisely reflects the caching decisions for the Spark application, e.g., which RDDs are persisted, and when they are persisted and unpersisted. Take Figure 2 as an example. After running this application, we can obtain an execution trace as follows: {*words.persist()*, *words.count()*, *result.persist()*, *result.count()*, *words.unpersist()*, *result.take(10)*, *result.unpersist()*}. From this trace, we can see that *words* is persisted before action *A*<sub>1</sub> (i.e., *words.count()*), and *words* is unpersisted after action *A*<sub>2</sub> (i.e., *result.count()*).

We collect the execution trace using dynamic method by slightly instrumenting Spark's code. The details are shown in Section 5. Thus, complex code structures, e.g., loops and recursions, will be unfolded as a sequential trace in this step.

### 4.2 Correct Caching Decision Generation

To judge whether the caching decisions in a given actual execution trace  $\tau_a$  are correct or not, we construct the correct execution trace  $\tau_c$ , which reflects the correct caching decisions. In order to do so, we extract all actions in the actual execution trace  $\tau_a$ , and ignore all caching decisions in  $\tau_a$ . Then, we construct the correct execution trace  $\tau_c$  in three steps. First, we identify all RDDs that need to be persisted (Section 4.2.1). Second, we identify the correct locations for persisting these RDDs (Section 4.2.2). Third, we identify the correct locations for unpersisting these RDDs (Section 4.2.3).

**4.2.1 Identify RDDs That Need to be Persisted.** As discussed in Section 3.1, if an RDD is used by multiple actions, the RDD needs to be persisted. In the lineage graph, the reused RDD is depended by multiple actions. For example, in Figure 2, *words* is used by action *A*<sub>1</sub> and *A*<sub>2</sub>. Thus, *words* should be persisted. However, not all RDDs that are depended by multiple actions should be persisted. In Figure 2, *pairs* is depended by action *A*<sub>2</sub> and *A*<sub>3</sub>. If we persist *pairs* and do not persist *result*, RDD *result* will be computed twice by *A*<sub>2</sub> and *A*<sub>3</sub>, respectively. If we persist *pairs* and *result*, the cached *pairs* will not be reused, and cause an unnecessary persist (Section 3.3). Therefore, CacheCheck needs to identify optimal decisions about what RDDs should be persisted.

**Algorithm 1:** Identify RDDs that need to be persisted.

```

Input: actions (Executed action list)
Output: cachedRDDs
1 cachedRDDs ← ∅;
2 foreach  $a_1, a_2 \in actions$  do
3   | identifyCachedRDDs ( $a_1.rdd, a_2$ );
4 end
5
6 Function identifyCachedRDDs ( $rdd, a_2$ ) do
7   | if  $a_2.dependsOn(rdd)$  then
8     |   | cachedRDDs.add( $rdd$ );
9   | else
10    |   | foreach pRDD ∈  $rdd.parentRDDs$  do
11      |   |   | identifyCachedRDDs ( $pRDD, a_2$ );
12    |   | end
13 end

```

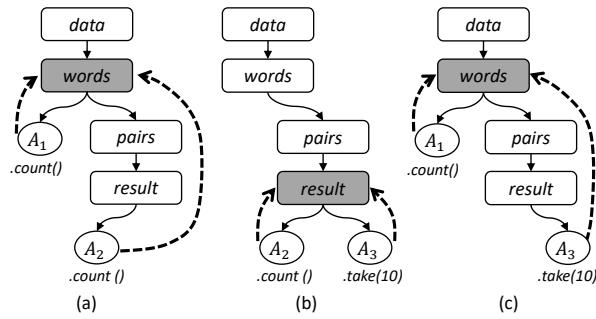


Figure 4: Identify cached RDDs for the example in Figure 2.

Algorithm 1 shows how to identify RDDs that need to be persisted for an execution trace. We aim to find the minimal set of RDDs, if they are persisted, all RDD recomputation will be eliminated. For every two actions  $a_1$  and  $a_2$  in the executed action list  $actions$ , CacheCheck finds the closest RDDs that are depended by both  $a_1$  and  $a_2$  (Line 2–4). CacheCheck starts from the RDD that  $a_1$  directly depends on ( $a_1.rdd$  at Line 3), and checks whether the RDD is used by action  $a_2$ . If  $a_2$  also depends on  $rdd$ , then  $rdd$  is one of the closest RDDs that are used by two actions, and  $rdd$  needs to be persisted (Line 7–8). If  $a_2$  does not depend on  $rdd$ , CacheCheck further checks whether the parent RDDs of  $rdd$  are depended by  $a_2$  (Line 10–12).

Take Figure 2 as an example. This example contains three actions:  $words.count()$ ,  $result.count()$  and  $result.take(10)$ . The cached RDD identification process is shown in Figure 4. For action  $words.count()$  and  $result.count()$ , the closest RDD shared by them is  $words$  as shown in Figure 4a. For action  $result.count()$  and  $result.take(10)$ , the closest RDD shared by them is  $result$  as shown in Figure 4b. For action  $words.count()$  and  $result.take(10)$ , the closest RDD shared by them is  $words$  as shown in Figure 4c. Therefore,  $words$  and  $result$  need to be persisted.

**4.2.2 Identify Persist Locations.** After finding all the RDDs that should be persisted in Algorithm 1, we need to identify when to

**Algorithm 2:** Generate correct caching decisions.

```

Input: actions (Executed action list), cachedRDDs
Output:  $\tau_c$  (Correct trace)
1  $\tau_c \leftarrow \emptyset$ ;
2  $\tau_c.addAll(actions)$ ;
3 /* Identify persist locations */
4 foreach rdd ∈ cachedRDDs do
5   | for  $i \leftarrow 1; i \leq actions.length; i++$  do
6     |   | if  $actions[i].dependsOn(rdd)$  then
7       |   |   |  $\tau_c.insertBefore(actions[i], rdd.persist())$ ;
8     |   | break;
9   | end
10 end
11 /* Identify unpersist locations */
12 foreach rdd ∈ cachedRDDs do
13   | lastAction ← NULL;
14   | for  $i \leftarrow 1; i \leq actions.length; i++$  do
15     |   | if use ( $action[i], actions[i].rdd, rdd$ ) then
16       |   |   | lastAction ←  $actions[i]$ ;
17     |   | end
18     |   |  $\tau_c.insertAfter(action[i], rdd.unpersist())$ ;
19   | end
20
21 /* Does curRDD use tarRDD? */
22 Function use ( $action, curRDD, tarRDD$ ) do
23   | if  $curRdd = tarRDD$  then
24     |   | return True;
25   | else if isCached ( $action, curRDD$ ) = False then
26     |   | foreach pRDD ∈  $curRDD.parentRDDs$  do
27       |   |   | if use ( $action, pRDD, tarRDD$ ) then
28         |   |   |   | return True;
29     |   | end
30   | return False;
31 end
32
33 /* Has rdd been cached before executing action? */
34 Function isCached ( $action, rdd$ ) do
35   | if cachedRDDs.contains( $rdd$ ) = False then
36     |   | return False
37   | preAction ←  $\tau_c.previousAction(action)$ ;
38   | if preAction ≠ NULL then
39     |   | if  $\tau_c.isBefore(rdd.persist(), preAction)$  then
40       |   |   | return True;
41     |   | return False;
42 end

```

persist these RDDs. Algorithm 2 presents how to identify when to persist an RDD. First, we construct an initial execution trace by adding all actions into it (Line 2). For the example shown in Figure 2, the initial execution trace is  $\{words.count(), result.count(), result.take(10)\}$ .

For each RDD  $rdd$  in  $cachedRDDs$ , the operation  $rdd.persist()$  should be inserted before the first action that depends on  $rdd$  (Line

**Algorithm 3:** Detect cache-related bugs.

---

**Input:**  $\tau_c$  (Correct trace),  $\tau_a$  (Actual trace)

```

1 for  $i \leftarrow 1; i \leq \tau_a.length; i + + \text{ do}$ 
2   |  $op \leftarrow \tau_c[i];$ 
3   | if  $op \notin \tau_a$  then
4     |   | if  $op.isPersist$  then
5       |     |   | Report 'missing persist on  $op.rdd$ ';
6     |   | if  $op.isUnpersist$  then
7       |     |   | if  $!op.isAfter(\tau_c.lastAction) \wedge$ 
8         |       |     |  $\tau_a.havePersist(op.rdd)$  then
9           |         |   | Report 'missing unpersist on  $op.rdd$ ';
10      |   | else
11        |     |   | if  $op.isPersist$  then
12          |       |     |  $cAction \leftarrow \tau_c.nextAction(op);$ 
13          |       |     |  $aAction \leftarrow \tau_a.nextAction(op);$ 
14          |       |     | if  $cAction.isBefore(aAction)$  then
15            |         |   | Report 'lagging persist on  $op.rdd$ ';
16        |     |   | else if  $op.isUnpersist$  then
17          |       |     |  $cAction \leftarrow \tau_c.previousAction(op);$ 
18          |       |     |  $aAction \leftarrow \tau_a.previousAction(op);$ 
19          |       |     | if  $cAction.isBefore(aAction)$  then
20            |         |   | Report 'lagging unpersist on  $op.rdd$ ';
21        |     |   | else if  $aAction.isBefore(cAction)$  then
22          |         |   | Report 'premature unpersist on  $op.rdd$ ';
23 end
24 for  $i \leftarrow 1; i \leq \tau_a.length; i + + \text{ do}$ 
25   |  $op \leftarrow \tau_a[i];$ 
26   | if  $op \notin \tau_c$  then
27     |   | if  $op.isPersist$  then
28       |         |   | Report 'unnecessary persist on  $op.rdd$ ';
end
```

---

4–10). For the example shown in Figure 2, *words* and *result* should be cached. For RDD *words*, *words.count()* is the first action that depends on it. For RDD *result*, *result.count()* is the first action that depends on it. Therefore the new trace that contains the RDD persist locations is:  $\{\text{words.persist}(), \text{words.count}(), \text{result.persist}(), \text{result.count}(), \text{result.take}(10)\}$ .

**4.2.3 Identify Unpersist Locations.** If a cached RDD is not used any more by following actions, it should be unpersisted as early as possible to save memory. That is to say, once the last action that uses a cached RDD has been executed, the RDD should be discarded. First, we find the last action which uses a cached RDD (Line 13–17). Then, we insert *rdd.unpersist()* right after the last action (Line 18). To decide if an *action* uses an RDD *rdd*, CacheCheck starts to check if the RDD *curRDD* that *action* directly depends on uses *rdd* (Line 15). If *curRDD* is not cached before, then CacheCheck checks whether its parent RDDs use *tarRDD* (Line 25–29). If *curRDD* has been cached (Line 34–42), then CacheCheck will stop search, since *action* will use cached *curRDD*, and does not use its parent RDDs. Take Figure 2 as an example. the last action that uses *words* is *result.count()* and the last action that uses *result* is *result.take(10)*. Note that, once *result* is persisted by action  $A_2$ , action  $A_3$  will use

the cached *result*, and will not use *words* any more. Therefore, the new correct trace is:  $\{\text{words.persist}(), \text{words.count}(), \text{result.persist}(), \text{result.count}(), \text{words.unpersist}(), \text{result.take}(10), \text{result.unpersist}()\}$ .

### 4.3 Bug Detection

CacheCheck detects cache-related bugs by analyzing the difference of caching decisions in the collected actual execution trace ( $\tau_a$ ) and the generated correct execution trace ( $\tau_c$ ) in Section 4.2. For all the buggy examples in Figure 3, CacheCheck can generate the same correct execution trace:  $\{\text{words.persist}(), \text{words.count}(), \text{result.persist}(), \text{result.count}(), \text{words.unpersist}(), \text{result.take}(10), \text{result.unpersist}()\}$ . However, their actual execution traces are different, and illustrate different cache-related bugs. Algorithm 3 shows our cache-related bug detection. We illustrate it as follows.

(1) Missing persist. If an operation *rdd.persist()* belongs to the correct execution trace but does not belong to the actual execution trace, CacheCheck reports a missing persist bug (Line 3–5).

(2) Lagging persist. Both the correct execution trace and the actual execution trace contains an operation *rdd.persist()*. However, in the actual execution trace, *rdd.persist()* is not performed before the first action that uses *rdd* (Line 10–14). Take Figure 3b as an example. We can get the actual execution trace as  $\{\text{words.count}(), \text{words.persist}(), \text{result.persist}(), \text{result.count}(), \text{words.unpersist}(), \text{result.take}(10), \text{result.unpersist}()\}$ . We can see that *words.persist()* is located before *words.count()* in the correct trace. However, it is located after *words.count()* in the actual execution trace. So, CacheCheck reports a lagging persist bug on RDD *words*.

(3) Unnecessary persist. If an operation *rdd.persist()* belongs to the actual execution trace but does not belong to the correct execution trace, CacheCheck reports an unnecessary persist bug (Line 25–27).

(4) Missing unpersist. If an RDD *rdd* that should be cached has been persisted in the actual execution trace, but the operation *rdd.unpersist()* belongs to the correct execution trace but not belong to the actual execution trace, and the operation is not located after the last action in the trace, CacheCheck reports a missing unpersist bug (Line 6–8).

(5) Premature unpersist. Both the correct execution trace and the actual execution trace contain an operation *rdd.unpersist()*. However, in the actual execution trace, *rdd.unpersist()* is performed before the last action that uses *rdd* (Line 20–21). Take Figure 3e as an example. The actual execution trace of this example is  $\{\text{words.persist}(), \text{words.count}(), \text{words.unpersist}(), \text{result.persist}(), \text{result.count}(), \text{result.take}(10), \text{result.unpersist}()\}$ . *words.unpersist()* is located after *result.count()* in the correct execution trace, while it is located before *result.count()* in the actual trace. Thus, CacheCheck reports a premature unpersist bug on RDD *words*.

(6) Lagging unpersist. Both the correct execution trace and the actual execution trace contain an operation *rdd.unpersist()*. However, in the actual execution trace, *rdd.unpersist()* is not performed right after the last action that uses *rdd* (Line 15–19).

## 5 IMPLEMENTATION

CacheCheck first collects an execution trace by modifying Spark implementation, and then performs cache-related bug detection offline

on the execution trace. Here, we mainly introduce the implementation details which are not described in Section 4, including code instrumentation for trace collection, and bug reports for helping inspect bugs and bug deduplication.

**Execution trace collection.** Spark invokes *SparkContext.runJob()* to execute each action in a Spark application. We modify *SparkContext.runJob()*, and record the information about each action, including the action ID, the RDD that the action works on, and the lineage graph of the action. To obtain each cache-related operation, we modify *RDD.persist()* and *RDD.unpersist()*, and record the RDD ID for each *persist()* and *unpersist()* operation. Our modification to the Spark system is quite lightweight, and involves only 338 lines of code. The overhead for obtaining execution traces is usually negligible (less than 5% in our experiments).

Note that, we modify the Spark system to collect execution traces of Spark applications. Therefore, developers only need to run their Spark applications on our modified Spark system, without instrumenting or modifying their applications in any way. CacheCheck can automatically collect their execution traces and perform offline analysis for cache-related bug detection.

**Bug reports.** CacheCheck detects cache-related bugs based on the execution trace. If a cache-related bug at the same code location is executed multiple times, CacheCheck can report the cache-related bug multiple times. In order to eliminate duplicate bug reports, we consider two bug reports are the same if they satisfy the following two conditions: (1) Their bug-related RDDs are generated in the same code locations. (2) They have the same bug information, e.g., bug pattern and code locations of cache-related API invocations.

To facilitate bug inspection for developers, we provide the following information for each reported bug: (1) The code positions and call stacks for each operation that relates to the bug report, e.g., where a related RDD is generated, where a cache-related API is called on an RDD, and invocation position for each action. (2) The lineage graph for each action that relates to the bug report. For example, for a missing persist on *rdd*, CacheCheck provides where two actions use *rdd*, and which action firstly uses *rdd*. For a premature unpersist on *rdd*, CacheCheck provides where the *unpersist()* is invoked on *rdd*, and actions that use *rdd* after the *unpersist()* operation.

## 6 EVALUATION

We evaluate CacheCheck and study the following three research questions:

**RQ1:** *Can cache-related bugs seriously affect the performance of Spark applications?*

**RQ2:** *Can CacheCheck effectively detect known cache-related bugs in Spark applications?*

**RQ3:** *Can CacheCheck detect unknown cache-related bugs in real-world Spark applications?*

To answer RQ1 and RQ2, we first build a cache-related bug benchmark with 18 bugs (Section 6.1). To answer RQ1, we compare the performance difference between the buggy versions and their corresponding fixed versions (Section 6.2). To answer RQ2, we evaluate CacheCheck on our bug benchmark (Section 6.3). To answer RQ3, we evaluate CacheCheck on six real-world Spark applications to validate if CacheCheck can detect new bugs (Section 6.4).

### 6.1 Cache-Related Bug Benchmark

We select the six bug examples in Figure 3 as our micro-benchmark (P1–P6 in Table 1). We further manually inspect issue reports in open source Spark applications, to collect real-world cache-related bugs. We select three representative Spark applications, i.e., GraphX [22], MLlib [23] and Spark SQL [29], as our study applications. These applications are representative official Spark applications, well-maintained, and widely used in practice. All issue reports in these applications are publicly available in Spark JIRA [28].

We use cache-related keywords, i.e., cache and persist, to search all issue reports in GraphX, MLlib and Spark SQL. Two authors and two master students carefully analyze each returned issue report. If an issue report satisfies the following conditions, we select it into our benchmark. (1) We can identify its root cause, and confirm that it is a cache-related bug. (2) We can find its fixing patch, and obtain its buggy version and fixed version. (3) We can reproduce it based on an existing test case in its application.

Through the above process, we finally obtain 12 cache-related bugs (B1–B12 in Table 1). Note that, these 12 cache-related bugs only cover three bug patterns discussed in Section 3, i.e., 3 missing persist bugs, 3 unnecessary persist bugs, and 6 missing unpersist bugs. We do not observe cache-related bugs for other three bug patterns, i.e., lagging persist, premature unpersist and lagging unpersist. However, as discussed in Section 6.4, we detect 15 bugs for these three bug patterns. Among them, 11 have been confirmed by developers, and 7 have been fixed. This indicates that developers may not have realized these three bug patterns before our study.

### 6.2 Performance Slowdown Caused by Cache-Related Bugs

**Experimental design.** We use the 18 cache-related bugs in our bug benchmark in Table 1 as our experimental subjects. We run this experiment in the local mode with 8GB memory. For the six cache-related bugs (P1–P6), we use a 660MB text file as input. For the 12 real-world cache-related bugs (B1–B12), we run their corresponding test cases with inputs of 23.4–708MB. For each cache-related bug, we run its buggy version and fixed version 10 times, and then use the average execution time to compare the performance slowdown.

**Experimental result.** The columns 5–7 in Table 1 (Execution time) show the absolute execution time of the buggy / fixed versions and the performance slowdown. We can see that cache-related bugs can cause non-negligible (0.1% – 51.6%) performance slowdown. Eight cache-related bugs introduce more than 5% performance slowdown: P1, P2, P5, B1, B2, B3, B5, B8. Specially, in bug B2, all the cached RDDs in a loop are not unpersisted, and waste huge memory, thus causing Out of Memory (OOM). After correctly unpersisting all RDDs in the loop, the application terminates normally.

Generally, missing persist (e.g., P1, B5), lagging persist (e.g., P2), and premature unpersist (e.g., P5) can usually cause much performance slowdown, since they can cause recomputation of related RDDs. However, not all missing persist bugs can cause serious performance slowdown, e.g., B6 and B7. This is because recomputing the unpersisted RDDs in these two cases is not time-consuming. Missing unpersist (e.g., B2, B3) and unnecessary persist (e.g., B8) can cause much performance slowdown sometimes, if they occupy too much memory.

**Table 1: Experimental Results on Known Bugs**

ID	Issue ID	App	Pattern	Execution time			Detected	New bugs
				Buggy(s)	Fixed(s)	Slowdown		
P1	Pattern-MP (Figure 3a)	Word count	Missing persist	31.7	26.1	21.4%	✓	0
P2	Pattern-LP (Figure 3b)	Word count	Lagging persist	32.1	26.1	22.9%	✓	0
P3	Pattern-UP (Figure 3c)	Word count	Unnecessary persist	27.3	26.1	4.5%	✓	0
P4	Pattern-MUP (Figure 3d)	Word count	Missing unpersist	26.3	26.1	0.9%	✓	0
P5	Pattern-PUP (Figure 3e)	Word count	Premature unpersist	35.4	26.1	35.7%	✓	0
P6	Pattern-LUP (Figure 3f)	Word count	Lagging unpersist	26.2	26.1	0.3%	✓	0
B1	SPARK-1266 [3]	MLlib	Unnecessary persist	24.7	23.4	5.2%	✓	4
B2	SPARK-3918 [6]	MLlib	Missing unpersist	OOM	27.2	-	✓	0
B3	SPARK-7100 [10]	MLlib	Missing unpersist	27.6	26.1	5.9%	✓	0
B4	SPARK-10182 [9]	MLlib	Missing unpersist	34.2	33.6	1.8%	✓	2
B5	SPARK-16697 [13]	MLlib	Missing persist	67.8	44.7	51.6%	✓	1
B6	SPARK-16880 [14]	MLlib	Missing persist	262.2	256.3	2.3%	✓	1
B7	SPARK-18356 [15]	MLlib	Missing persist	20.9	20.7	0.7%	✓	2
B8	SPARK-18608 [16]	MLlib	Unnecessary persist	244.9	198.7	23.3%	✓	3
B9	SPARK-26006 [20]	MLlib	Missing unpersist	56.0	54.8	2.0%	✓	2
B10	SPARK-2661 [4]	GraphX	Missing unpersist	13.7	13.1	4.6%	✓	0
B11	SPARK-3290 [5]	GraphX	Missing unpersist	11.9	11.7	1.8%	✓	8
B12	SPARK-7116 [11]	Spark SQL	Unnecessary persist	20.4	20.2	0.1%	✓	0

### 6.3 Detecting Known Cache-Related Bugs

We use the 18 cache-related bugs in Table 1 to evaluate the cache-related bug detection ability of CacheCheck. For each cache-related bug in Table 1, we run its test case, and use CacheCheck to detect cache-related bugs on the collected execution trace.

The last two columns in Table 1 show the detection result. We can see that CacheCheck can detect all 18 cache-related bugs (Detected). CacheCheck further detects 23 new cache-related bugs (New bugs) in these test cases. Two authors and two master students manually inspect these new detected cache-related bugs to validate whether they are real bugs. After manual validation, we find that all these newly detected cache-related bugs are true. Since these new cache-related bugs are detected on old versions of GraphX, MLlib and Spark SQL, we do not submit them to the developers for further confirmation.

### 6.4 Detecting Unknown Cache-Related Bugs

In this section, we evaluate CacheCheck on six real-world Spark applications, and obtain developers' feedbacks about our detected cache-related bugs.

**6.4.1 Experimental Design.** We collect six Spark applications for our experiments from two aspects. First, we use GraphX [22], MLlib [23], Spark SQL [29], which are the three official Spark applications, and well maintained by the Spark community. Second, we collect another three Spark applications from GitHub by the following steps. (1) We used the keyword “Apache Spark” to search GitHub repositories written in Scala, and obtained around 1600 repositories. (2) We sorted these repositories by their stargazers in descending order, and manually checked them one by one. (3) We aimed to select representative data process applications. Thus, If a repository is used for examples, learning, and extensions of Spark framework,

we ignored it. (4) If a repository contains test cases, and we can run them on Spark 2.4.3, then we selected it. By following the above steps, we obtained three applications and evaluated CacheCheck on them. The first four columns in Table 2 show the basic information about all these six Spark applications.

We perform our experiments as follows. First, we run the test cases included in these applications on our modified Spark implementation (Section 5) and collect their execution traces. The fifth and sixth columns in Table 2 show the numbers of collected actions and RDDs in the execution traces. Second, we use CacheCheck to detect cache-related bugs based on these execution traces.

In our experiment, a cache-related bug may be triggered multiple times by different test cases. We adopt the approach discussed in Section 5 to remove duplicated bug reports. When counting bugs, we only count unique bugs, i.e., remove duplicated bug reports from different test cases. MCL [8] and t-SNE [7] are built on MLlib. For these two applications, if we detect a bug occurring in MLlib, we only count it once in MLlib, and do not count it in these applications.

**6.4.2 Detection Result.** Table 2 shows the unique cache-related bugs detected by CacheCheck. In total, CacheCheck detects 72 cache-related bugs, covering the six cache-related bug patterns in Section 3. In detail, CacheCheck detects 34 missing persist bugs (MP), 1 lagging unpersist bug (LP), 18 unnecessary persist bugs (UP), 5 missing unpersist bugs (MUP), 12 lagging unpersist bugs (LUP), and 2 premature unpersist bugs (PUP). We can see that cache-related bugs are common in real-world Spark applications.

We have submitted all these 72 bugs to developers through JIRA and GitHub. So far, 58 bugs have been discussed by developers, and 53 of them have been confirmed as real bugs. The remaining 5 bugs are not decided yet. Developers have fixed 28 bugs. Among the 14 bugs that have not received responses, 9 occur in third-party

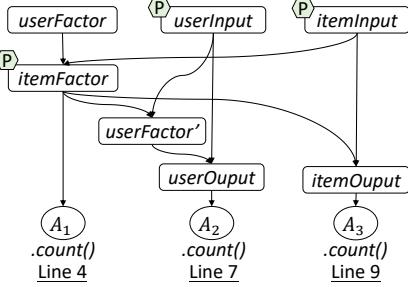
**Table 2: Detecting Cache-Related Bugs on Real-World Spark Applications**

App	Description	Star	LOC	Action	RDD	MP	LP	UP	MUP	LUP	PUP
GraphX [22]	Graph computation	-	32,708	11	236	1(1)	0	6(5)	1(1)	0	0
MLlib [23]	Machine learning	-	109,786	888	4,017	24(21*)	1(1)	10(10)	2(1)	12(8)	2(2)
Spark SQL [29]	SQL	-	395,122	69	263	4(3)	0	0	0	0	0
MCL [8]	Markov clustering	32	1,058	48	290	4	0	0	0	0	0
Betweenness [12]	k betweenness centrality	40	488	6	83	0	0	1	0	0	0
t-SNE [7]	Distributed t-SNE	129	798	39	97	1	0	1	2	0	0
<b>Total</b>				1,061	4,986	34(25)	1(1)	18(15)	5(2)	12(8)	2(2)

\* The numbers in the parentheses denote the bugs confirmed by developers.

```

1: userInput.persist();
2: itemInput.persist();
3: itemFactor = computeFacotrs(userFactor, itemInput).persist()
4: itemFactor.count()
5: userFactor = computeFactors(itemFactor, userInput)
6: val userOutput = userInput.join(userFactor)
7: val itemOutput = itemInput.join(itemFactor)
8: userOutput.count()
9: itemOutput.count()
10: userInput.unpersist() // Premature unpersist
11: itemOutput.unpersist()
12: itemInput.unpersist()
```



**Figure 5: Premature unpersist and lagging unpersist bugs in simplified ALS algorithm in SPARK-29844 [24]. *userFactor'* denotes a new RDD created in Line 5, which is different from the original RDD *userFactor* in Line 3.**

applications on GitHub, e.g., MCL [21], mainly due to the inactive maintenance of these applications.

Note that, we have detected 1 lagging persist bugs, 12 lagging unpersist bugs, and 2 premature unpersist bugs in real-world Spark applications. Among them, 11 have been confirmed by developers, and 7 have been fixed. Although we did not observe these bug patterns when we collecting known cache-related bugs, developers confirm that these bug patterns are harmful.

**6.4.3 Feedbacks from Developers.** So far, developers have given feedbacks on 58 cache-related bugs. In the process of bug submission, developers have shown great interests in CacheCheck. For example, after they confirmed and resolved one bug [24], they gave the following comments to encourage us to submit more cache-related bugs: “*Nice, I wonder what else CacheCheck will turn up.*”

Besides, we also received some interesting comments from developers, which indicates that even experienced developers may not be easy to figure out complicated caching decisions. For example, Figure 5 shows the simplified code of our bug report SPARK-29844 [24] in MLlib. There are two cache-related bugs in this report. (1) *itemFactor* (Line 8) should be unpersisted after *itemOutput.count()* (Line 9), instead of after *userOutput.count()* (Line 7). This causes a premature unpersist bug. (2) *userInput* (Line 10) should be unpersisted before *itemOutput.count()* (Line 9), instead of after *itemOutput.count()* (Line 9). This causes a lagging unpersist bug. In this bug, a developer left the following comment in its PR [25]: “*I don't see the problem here immediately. Doesn't this (itemOutput) depend on some of the cached user RDDs (userInput)?*” The developer thought

```

1: baggedInput.persist()
2: while(treeNode.nonEmpty) {
3:   // findBestSplits will invoke an action
4:   RandomForest.findBestSplits(baggedInput)
5: }
6: baggedInput.unpersist()
```

**Figure 6: An unnecessary persist bug when Line 4 in the while statement is executed only once in MLlib [26].**

*itemOutput* depends on *userInput*, so *userInput* should be unpersisted after all actions completed. After we explained the fact that *itemOutput* only depends on the cached *itemFactor*, he agreed that moving *userInput.unpersist()* before *itemOutput.count()* is a kind of optimization.

**Why developers do not fix some cache-related bugs?** Among the 53 confirmed cache-related bugs, developers do not fix 25 of them for now. We summarize the reasons why they do not fix them as follows.

(1) Developers thought that the performance slowdown caused by cache-related bugs should be negligible. 17 bugs belong to this case. First, the should-be-persisted RDD is small, and the overhead of recomputation is negligible, e.g., SPARK-29872 [27]. Second, the bugs can only occur in rare cases. For example, Figure 6 shows an unnecessary persist bug in MLlib [26]. If the iteration (Line 2–5) is only executed once, the persist on RDD *baggedInput* is unnecessary. Developers thought this rarely occurs in the production environment. Third, the bugs occur in illustrative code (i.e., code

```

1: def apply(vertices, edges): Graph[V, E] = {
2:   vertices.persist()
3:   val newEdges = edges.map(part => part.withoutVertex).persist()
4:   new Graph(vertices, newEdges)
5: }

```

**Figure 7: An API related to cache decisions in GraphX [17].**

examples), and developers thought fixing the bugs can make code understanding difficult, e.g., SPARK-29872 [27].

(2) Developers thought that fixing cache-related bugs makes APIs difficult to utilize. 8 bugs belong to this case. Spark applications, e.g., GraphX, MLLib and Spark SQL, can be used as APIs to build other applications. The caching decisions of these APIs may not consider all their usage scenarios. Figure 7 shows an unnecessary persist bug in GraphX [17]. In API *apply()*, RDD *vertices* and *newEdges* are persisted. This API can be invoked by many graph processing algorithms in GraphX, e.g., SVDPlusPlus and PageRank. Both *vertices* and *newEdges* are used by only one action in SVD-PlusPlus, so CacheCheck reports two unnecessary persist bugs on them. However, in PageRank, they are used by multiple actions in an iterative computation, so the persist decisions are correct. Developers wanted to simplify the usage of these APIs, and would not like to fix them for different applications.

**6.4.4 Slowdowns of Detected Unknown Bugs.** The slowdowns of cache-related bugs are usually highly related to the size of input data in the Spark applications. We detected unknown cache-related bugs by running test cases included in the applications. Note that, these test cases usually run on small amount of data (e.g., a string with 100 characters, a graph with 10 nodes), and complete quickly (e.g., in less than 1 second). In these scenarios, the slowdowns can be ignored. Therefore, we did not report the slowdowns for these test cases. However, for real scenarios, these test cases will be run on large amount of data (e.g., 100G) for a long time (e.g., 30 minutes). In these scenarios, the slowdowns will be greatly important. Thus, for the newly reported bugs that have been fixed, we run their test cases with large amount of data like what we did in Section 6.2, and analyzed their slowdowns. Our experiment shows that, the slowdowns range from 0.2% to 35.4%, and 3 bugs cause Out of Memory (OOM).

## 7 DISCUSSION

While our experiments show that CacheCheck is promising in detecting cache-related bugs in Spark applications, we discuss potential threats and limitations of our work.

**Representativeness of our studied Spark applications.** We select a number of cache-related bugs and Spark applications in our experiments. Our selected cache-related bugs come from real-world Spark applications. Our selected Spark applications, e.g., GraphX [22], MLLib [23] and Spark SQL [29], are well maintained and actively developed by the Spark community. Thus, we believe our studied cache-related bugs and Spark applications can represent the real-world cases.

**Trade-off between memory and performance.** Whether an RDD should be persisted or not is usually a trade-off between execution performance and the memory requirement of the persisted

RDD. Our approach does not take the memory requirement into consideration. However, we believe that CacheCheck can still give developers useful suggestions about how to enforce proper caching decisions. Additionally, Spark provides various cache storage levels, e.g., off-heap, on-disk, in-memory. CacheCheck cannot provide useful suggestions for cache levels.

**Generalization to other systems.** CacheCheck is currently designed for Spark. It may be generalized to other big data systems that have the same or similar cache mechanisms as Spark. For example, in order to reduce the cost of recomputation and reloading files, Hadoop and Flink have utilized or planned to utilize similar cache mechanisms. CacheCheck may be adapted into these systems.

## 8 RELATED WORK

To the best of our knowledge, no previous work can detect cache-related bugs in Spark applications. In this section, we discuss related work that is close to ours.

**Cache optimization in Spark.** Existing work mainly focuses on cache optimization techniques in Spark implementation. Neutrino [47] employs fine-grained memory caching of RDD partitions and uses different in-memory cache levels based on runtime characteristics. Yang et al. [48] developed some adaptive caching algorithms to make online caching decisions with optimality guarantees. Zhang et al. [52] studied the influence of disk use in the Spark cache mechanism, and proposed a method of combined use of memory and disk caching. These works may eliminate the use of *persist()* and *unpersist()* operations and cache levels. To meet this all-or-nothing property in data access, PACMan [30] coordinates access to the distributed caches and designs new cache replacement mechanisms. LRC (Least Reference Count) [49] exploits the application-specific DAG information to optimize the cache replacement mechanism. However, it is still a common practice to enforce caching decisions by developers causing severe performance slowdown. CacheCheck automatically detects improper *persist()* and *unpersist()* usage in Spark applications. Thus, our CacheCheck is orthogonal to these pieces of existing work.

**Memory management optimization in Spark.** Researchers have proposed some approaches to improve memory management in big data systems. An experimental study [46] has shown that garbage collectors can affect big data processing performance. Yak [39] divides the managed heap in JVM into a control space and a data space, and optimizes garbage collection for big data systems. Skyway [38] can directly connect managed heaps of different (local or remote) JVM processes, and optimizes object serialization / deserialization. Stark [36] optimizes in-memory computing on dynamic collections. Panthera [42] analyzes user programs running on Apache Spark to infer their coarse-grained access patterns, and then provides fully automated memory management technique over hybrid memories. Different from these approaches, CacheCheck focuses on the caching decisions in Spark applications.

**Resource leak detection in traditional programs.** Resource leak in programs can cause severe problems, e.g, performance degradation and system crash [44]. Researchers have made lots of efforts to detect resource leak. Sigmund et al. [32] and Yulei et al. [40] utilized value-flow analysis to detecting memory leaks in C programs. Emina et al. [41] developed Tracker to detect resource leak in Java

programs. Relda [34] and Relda2 [44] detect resource leak in Android applications. However, all these works cannot handle with the execution model of Spark, e.g., lazy execution. Thus, they cannot be applied to detect cache-related bugs in Spark applications.

## 9 CONCLUSION

Apache Spark uses caching mechanisms to reduce the RDD recomputation in Spark applications, and relies on developers to enforce correct caching decisions. In this paper, we summarize six cache-related bug patterns in Spark applications. Our experiments on real-world cache-related bugs show that cache-related bugs can seriously degrade the performance of Spark applications. We further propose CacheCheck, which can automatically detect cache-related bugs by analyzing the execution traces of Spark applications. Our evaluation on real-world Spark applications shows that CacheCheck can effectively detect cache-related bugs. In the future, we plan to further improve the detection capability of CacheCheck using new techniques, e.g., static analysis, and performance estimation of cache-related bugs.

## ACKNOWLEDGEMENTS

We thank Kai Zhang, Yange Fang, Kai Kang, and Xingtong Ye for their contributions in validating detected unknown bugs. This work was partially supported by National Key Research and Development Program of China (2017YFB1001804), National Natural Science Foundation of China (61732019, 61802377, 61702490), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

## REFERENCES

- [1] 2004. Apache Hadoop. Retrieved January 10, 2020 from <http://hadoop.apache.org/>
- [2] 2014. Mining ecommerce graph data with Apache Spark at Alibaba Taobao. Retrieved January 6, 2020 from <https://databricks.com/blog/2014/08/14/mining-graph-data-with-spark-at-alibaba-taobao.html>
- [3] 2014. SPARK-1266: Persist factors in implicit ALS. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-1266>
- [4] 2014. SPARK-2661: Unpersist last RDD in bagel iteration. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-2661>
- [5] 2014. SPARK-3290: No unpersist calls in SVDPlusPlus. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-3290>
- [6] 2014. SPARK-3918: Forget Unpersist in RandomForest.scala(train Method). Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-3918>
- [7] 2015. Distributed t-SNE via Apache Spark. Retrieved January 10, 2020 from <https://github.com/saurfang/spark-tsne/>
- [8] 2015. MCL spark. Retrieved January 10, 2020 from [https://github.com/joandre/MCL\\_spark/](https://github.com/joandre/MCL_spark/)
- [9] 2015. SPARK-10182: GeneralizedLinearModel doesn't unpersist cached data. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-10182>
- [10] 2015. SPARK-7100: GradientBoostTrees leaks a persisted RDD. Retrieved January 23, 2020 from <https://issues.apache.org/jira/browse/SPARK-7100>
- [11] 2015. SPARK-7116: Intermediate RDD cached but never unpersisted. Retrieved January 23, 2020 from <https://issues.apache.org/jira/browse/SPARK-7116>
- [12] 2015. Spark betweenness. Retrieved January 10, 2020 from <https://github.com/dmarcus/spark-betweenness/>
- [13] 2016. SPARK-16697: Redundant RDD computation in LDAOptimizer. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-16697>
- [14] 2016. SPARK-16880: Improve ANN training, add training data persist if needed. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-16880>
- [15] 2016. SPARK-18356: KMeans should cache RDD before training. Retrieved January 23, 2020 from <https://issues.apache.org/jira/browse/SPARK-18356>
- [16] 2016. SPARK-18608: Spark ML algorithms that check RDD cache level for internal caching double-cache data. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-18608>
- [17] 2016. SPARK-29878: Improper cache strategies in GraphX. Retrieved January 19, 2020 from <https://issues.apache.org/jira/browse/SPARK-29878>
- [18] 2017. Tuning Apache Spark for Large-Scale Workloads. Retrieved January 10, 2020 from <https://databricks.com/session/tuning-apache-spark-for-large-scale-workloads>
- [19] 2018. Moving eBay's Data Warehouse Over to Apache Spark – Spark as Core ETL Platform at eBay. Retrieved January 10, 2020 from <https://databricks.com/session/moving-ebays-data-warehouse-over-to-apache-spark-as-core-etl-platform-at-ebay>
- [20] 2018. SPARK-26006: mllib Prefixspan. Retrieved January 23, 2020 from <https://issues.apache.org/jira/browse/SPARK-26006>
- [21] 2019. Cache missing in MCL.scala. Retrieved January 25, 2020 from [https://github.com/joandre/MCL\\_spark/issues/20](https://github.com/joandre/MCL_spark/issues/20)
- [22] 2019. GraphX | Apache Spark. Retrieved January 10, 2020 from <http://spark.apache.org/graphx/>
- [23] 2019. MLlib | Apache Spark. Retrieved January 10, 2020 from <http://spark.apache.org/mllib/>
- [24] 2019. SPARK-29844: Improper unpersist strategy in ml.recommendation.ASL.train. Retrieved January 10, 2020 from <https://issues.apache.org/jira/browse/SPARK-29844>
- [25] 2019. SPARK-29844 pull request: Improper unpersist strategy in ml.recommendation.ASL.train. Retrieved January 10, 2020 from <https://github.com/apache/spark/pull/26469>
- [26] 2019. SPARK-29856: Conditional unnecessary persist on RDDs in ML algorithms. Retrieved January 25, 2020 from <https://issues.apache.org/jira/browse/SPARK-29856>
- [27] 2019. SPARK-29872 pull request: Improper cache strategy in examples. Retrieved January 10, 2020 from <https://github.com/apache/spark/pull/26498>
- [28] 2019. Spark JIRA. Retrieved January 10, 2020 from <https://issues.apache.org/jira/projects/SPARK>
- [29] 2019. Spark SQL | Apache Spark. Retrieved January 10, 2020 from <http://spark.apache.org/sql/>
- [30] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 267–280.
- [31] Mariem Brahem, Stephane Lopes, Laurent Yeh, and Karine Zeitouni. 2016. Astro-Spark: Towards a distributed data server for big data in astronomy. In *Proceedings of SIGSPATIAL PhD Symposium*. 3:1–3:4.
- [32] Sigmund Cherem, Lonnie Princehouse, and Radu Rugină. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 480–491.
- [33] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 599–613.
- [34] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 389–398.
- [35] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-fast big data analysis*. O'Reilly Media, Inc.
- [36] Shen Li, Md Tanvir Amin, Raghu Ganti, Mudhakar Srivatsa, Shanhai Hu, Yiran Zhao, and Tarek Abdelzaher. 2017. Stark: Optimizing in-memory computing for dynamic dataset collections. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*. 103–114.
- [37] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.
- [38] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 56–69.
- [39] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 349–365.
- [40] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 254–264.
- [41] Emilia Torlak and Satish Chandra. 2010. Effective interprocedural resource leak detection. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*. 535–544.
- [42] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of*

- ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 347–362.
- [43] Kun Wang, Ke Zhang, and Chengxue Gao. 2015. A new scheme for cache optimization based on cluster computing framework Spark. In *Proceedings of International Symposium on Computational Intelligence and Design (ISCID)*. 114–117.
- [44] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering (TSE)* 42, 11 (2016), 1054–1076.
- [45] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of International Conference on Management of Data (SIGMOD)*. 1071–1085.
- [46] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An experimental evaluation of garbage collectors on big data applications. *Proceedings of the VLDB Endowment (VLDB)* 12, 5 (2019), 570–583.
- [47] Mohit Xu, Erci andSaxena and Lawrence Chiù. 2016. Neutrino: Revisiting memory caching for iterative data analytics. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 16–20.
- [48] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. 2018. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *Proceedings of International Conference on Cloud Computing (CLOUD)*. 277–284.
- [49] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Lettaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*. 1–9.
- [50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2:1–2:14.
- [51] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *Proceedings of USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 10:1–10:7.
- [52] Kaihui Zhang, Yusuke Tanimura, Hidemoto Nakada, and Hirotaka Ogawa. 2017. Understanding and improving disk-based intermediate data caching in Spark. In *Proceedings of IEEE International Conference on Big Data (Big Data)*. 2508–2517.

# Patch Based Vulnerability Matching for Binary Programs

Yifei Xu\*

School of Software Engineering,  
MoE KLINNS  
Xi'an Jiaotong University  
China  
xyf0921@stu.xjtu.edu.cn

Fu Song

School of Information Science and  
Technology  
ShanghaiTech University  
China  
songfu@shanghaitech.edu.cn

Zhengzi Xu\*

School of Computer Science and  
Engineering  
Nanyang Technological University  
Singapore  
xu0002zi@e.ntu.edu.sg

Yang Liu

Nanyang Technological University  
Singapore  
Institute of Computing Innovation  
Zhejiang University  
China  
yangliu@ntu.edu.sg

Bihuan Chen

School of Computer Science  
Fudan University  
China  
bhchen@fudan.edu.cn

Ting Liu

School of Cyber Science and  
Engineering, MoE KLINNS  
Xi'an Jiaotong University  
China  
tingliu@mail.xjtu.edu.cn

## ABSTRACT

The binary-level function matching has been widely used to detect whether there are 1-day vulnerabilities in released programs. However, the high false positive is a challenge for current function matching solutions, since the vulnerable function is highly similar to its corresponding patched version. In this paper, the Binary X-Ray (BINXRAY), a patch based vulnerability matching approach, is proposed to identify the specific 1-day vulnerabilities in target programs accurately and effectively. In the preparing step, a basic block mapping algorithm is designed to extract the signature of a patch, by comparing the given vulnerable and patched programs. The signature is represented as a set of basic block traces. In the detection step, the patching semantics is applied to reduce irrelevant basic block traces to speed up the signature searching. The trace similarity is also designed to identify whether a target program is patched. In experiments, 12 real software projects related to 479 CVEs are collected. BINXRAY achieves 93.31% accuracy and the analysis time cost is only 296.17ms per function, outperforming the state-of-the-art works.

## CCS CONCEPTS

- Theory of computation → Program analysis; • Security and privacy → Software reverse engineering.

## KEYWORDS

Vulnerability Matching, Patch Presence Identification, Binary Analysis, Security

\*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397361>

## ACM Reference Format:

Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch Based Vulnerability Matching for Binary Programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397361>

## 1 INTRODUCTION

Vulnerability whose patch has been released is called as 1-day vulnerability. It would be exploited to attack the users who fail to adopt the latest security patches. It is one of the most serious and common security threats. The binary-level code matching has been considered as a good solution to detect 1-day vulnerabilities in released programs [14, 20, 46]. It compares the similarity between functions with known vulnerabilities and target functions in a given binary executable. If a target function is similar to a known vulnerable function, it will be predicated as vulnerable.

To improve the vulnerability detection capability, many works have been proposed to improve the binary-level code matching accuracy. DiscovRE [18] and CACOMPARE [23] achieve function matching across architectures by lifting the binary instructions to a unified intermediate representation. BLEX [17] uses program execution to extract the semantic features to improve the matching accuracy. BinGo [11] and BinGo-E [48] combines syntactic, structural and semantic features to produce more accurate matching results. However, it is difficult for the current function matching solutions to differentiate vulnerable and patched functions, since patches usually introduce subtle changes to fix vulnerabilities [37]. The patched functions would be identified as vulnerable, resulting in high false positive rates in detecting vulnerabilities [34]. As a result, these works require security experts to manually analyze potential vulnerable functions to find the genuine ones, which is time-consuming.

It is not trivial to address this problem. On the one hand, approaches need to be tolerant enough to identify vulnerable functions even with the presence of vulnerability-irrelevant small code changes like function upgrades and compiler optimizations. On the other hand, approaches also need to be precise enough to filter out

those already patched functions. Zhang and Qian [49] proposed an algorithm to determine whether a function has been patched or not. They extract syntactic and semantic changes from source code and build a “source code to binary” matching model. However, it needs to analyze the source code, and thus it is not applicable when the source code is not available. To the best of our knowledge, there lacks an effective and efficient approach for binary-level vulnerability matching with patch identification. We have summarized the following three properties for such an approach to be practical for real-world projects.

- **P1.** The approach needs to be accurate in identifying the patches in the target functions.
- **P2.** The approach needs to be scalable for large real-world programs.
- **P3.** The approach should use no information from the source code to work in closed source program binaries.

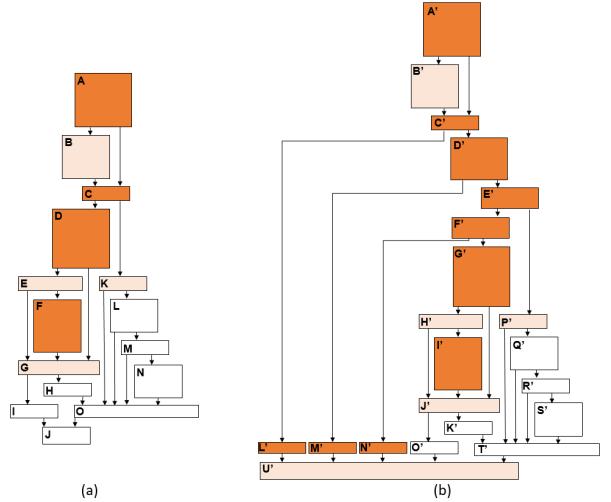
To fulfill these properties, we propose a patch based vulnerability matching approach, named as Binary X-Ray (BINXRAY). It can precisely differentiate patched functions from vulnerable functions in the binaries. It reduces the error rate by more than 30% compared to the state-of-the-art function matching tool, Bingo-E [48], with less time consumed. It is more accurate than the patch identification tool FIBER[49], without source code.

**For P1.** To accurately identify patched functions and detect real vulnerable functions, BINXRAY introduces two-step signature matching approach. First, to narrow the searching space, BINXRAY generates the function signature from the functions with known vulnerabilities, and uses the signatures to search for suspicious target functions in the binaries through matching. Second, it generates patch signatures by comparing the differences between the vulnerable functions and their patched versions. The patch signatures will be used to identify the patched functions from the suspicious target functions. To extract accurate patch signatures, we propose a structural basic block mapping algorithm to locate the changed and unchanged basic blocks between two functions. BINXRAY makes patch prediction based on the length sensitive similarity matching of the patch signature with the target function.

**For P2.** To improve the scalability, BINXRAY proposes patch signature extraction algorithm, which only captures essential parts of the patching semantics. Since most of the security patches only induce small changes within a few basic blocks in binary programs [29, 42, 50], BINXRAY locates the areas which only consist of the changes induced by patching vulnerabilities, and generates patch signatures based on them instead of generating signatures at the granularity of the whole function. This can also greatly improves the matching speed. Moreover, changes in other parts of the functions won’t be included in signatures, which are irrelevant to vulnerabilities. Thus, they will not affect the predicting results, leading to a more robust prediction against the noises from other changes.

**For P3.** From signature generation to patch identification, BINXRAY performs all the analysis at binary level. Therefore, it does not need any source code information which makes it suitable for programs without source code, such as firmware or third-party libraries.

This work makes the following contributions.



**Figure 1: Running Example: control flow graphs of the function `dtls1_process_heartbeat()` with the HeartBleed Bug in OpenSSL versions 1.0.1f (a) and 1.0.1g (b)**

- We propose a basic block mapping algorithm to accurately and efficiently map blocks in function differencing to generate patch signatures.
- A basic block boundary based algorithm is designed to locate the changes induced by patching. By reducing the size of patch signatures, it can improve the speed and scalability.
- We implement the prototype of BINXRAY, which can automatically extract patch signatures and accurately identify patched functions without any source code information. In experiments, 12 real software projects related to 479 CVEs are collected. BINXRAY achieves 93.31% accuracy in predicting the patch presence, and the analysis time cost is only 296.17ms per function, outperforming the state-of-the-art works.

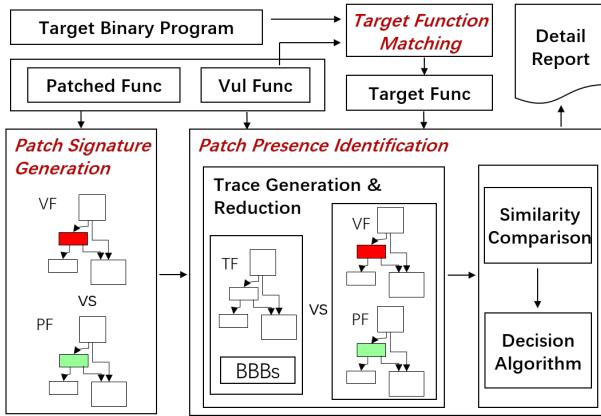
## 2 OVERVIEW

In this section, we first introduce a motivating example, and then present the overview of our approach.

### 2.1 Motivating Example

We use the HeartBleed bug (CVE-2014-0160 [1]) in OpenSSL as the running example. It occurs in `dtls1_process_heartbeat()` function. Figure 1(a) shows the control flow graph (CFG) of the vulnerable function in version 1.0.1f and Figure 1(b) shows the CFG of the patched function in version 1.0.1g. The patch adds a check in the source code which results in six new basic blocks in binary, named  $E'$ ,  $F'$ ,  $G'$ ,  $L'$ ,  $M'$  and  $N'$  in Figure 1(b). In addition, due to patching, the instructions in another four basic blocks, named  $A'$ ,  $C'$ ,  $D'$  and  $I'$  in Figure 1(b), are also slightly changed.

Given an unknown binary executable which may contain this function, we want to know whether the HeartBleed bug exists or not. If a traditional function matching approach is adopted and the vulnerable function in Figure 1(a) is used as the matching target, the corresponding function in the unknown binary may be matched. For instance, the patched function in Figure 1(b) can be matched,



**Figure 2: Overview of BINXRAY**

since they share a large number of common blocks  $B'$ ,  $H'$ ,  $P'$ ,  $J'$ ,  $K'$ ,  $O'$ ,  $Q'$ ,  $R'$ ,  $S'$ ,  $T'$  and  $U'$  with the same structure. However, it is unclear whether the matched function has been patched or not, since the two functions have a high degree of similarity, so tedious manual examination is usually necessary to differentiate genuine and spurious one.

BINXRAY is designed to address this problem. Its goal is to differentiate patched functions from vulnerable functions by identifying patch presences so that vulnerable functions can be identified with a low false positive rate.

## 2.2 Approach Overview

Figure 2 shows the overview of our approach, named BINXRAY. Taking the binary code of a vulnerable function (VF), a patched version of the same function, called patched function (PF), and a target program as inputs, the goal of BINXRAY is to effectively and efficiently check whether the target program has any functions (TF) that are similar to the vulnerable function but have not been patched yet. For each CVE, VF is the function before the patch commit. PF is the function after it. By diffing VF and PF, BinXray can generate the patch signature. TF is detected by the function matching algorithm using VF as the signature in the target binary. The core components of BINXRAY are: target function matching (Section 3.1), patch signature generation (Section 3.2), and patch presence identification (Section 3.3).

**Terms and Acronyms.** For convenient reference, we summarize the frequently used terms and their acronyms in Table 1.

**Target Function Matching.** BINXRAY generates lightweight function signatures and leverages function matching algorithm to quickly identify target functions (TFs) that are similar to the VF. Note that these TFs may not have been patched. The matching algorithm uses syntactic and structural information of VF, as the function signature, to identify TFs. Hence it is scalable and accurate enough to identify TFs.

**Patch Signature Generation.** BINXRAY automatically generates binary level patch signatures from the normalized binary code of the given VF and PF. Instead of incorporating the entire function into the signature, it first creates a mapping between the basic blocks (BBs) of two functions, from which BINXRAY identifies Changed

**Table 1: Terms and Acronyms**

Term	Acronym	Description
Vulnerable Function	VF	A function that contains a vulnerability
Patched Function	PF	A function whose vulnerability has been patched
Target Function	TF	A function to be checked whether the vulnerability has been patched or not
Basic Block	BB	A sequence of consecutive instructions without any branching
Changed Basic Block	CBB	A block that has been changed, added, or deleted in the differences between a VF and its PF
Boundary Basic Block	BBB	A block that is the parent or the child of a CBB, or the root or leave of a CFG
Trace	-	A sequence of consecutive BBs without any loops
Valid Trace	VT	A sequence of consecutive basic blocks that starts and ends with some BBBs, crosses at least one CBB, without any loops

Basic Blocks (CBBs) and computes two sets of valid traces (VTs): one set  $T_1$  from the CBBs of the VF and one set  $T_2$  from the CBBs of the PF. The two trace sets ( $T_1, T_2$ ) are regarded as a patch signature.

**Patch Presence Identification.** BINXRAY determines whether each identified TF has been patched or not, by matching it with the patch signature. If the TF is more close to VF than PF, it is considered to be vulnerable. Otherwise, it is considered to be patched.

### 3 METHODOLOGY

In this section, we elaborate the core components of our approach.

### 3.1 Target Function Matching

Given a target program, a VF and its PF, in order to reduce the time consumption, we first narrow down the searching scope by locating TFs in the target program that are similar to VF, on which the patch presence identification is performed. We adopt a lightweight technique for target function matching. For scalability consideration, taking insights from [11, 18, 20], we use syntactic and structural information of functions to construct the function signatures and check whether two functions are similar or not. The syntactic information consists of the sequence of mnemonic operators in binary instructions, function calls and constant values. The structural information consists of the number of instructions, basic blocks, branches and the control flow graph.

Using the function signatures, our target function matching achieves a high accuracy to identify the TFs. This is because that vulnerable functions are usually large whose syntactic and structural information are rich and unique, which allows us to differentiate them from other functions. Therefore, although the lightweight technique sacrifice a little accuracy in order to be scalable, the results are still accurate enough for patch presence identification in our experiments. Remark that this target function matching is not the contribution of this work, hence it may be similar to other existing approaches. We include it because we would like to show the complete workflow of our framework.

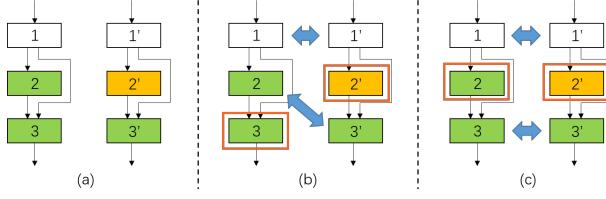


Figure 3: Case for Duplicate Basic Blocks Mapping

### 3.2 Patch Signature Generation

In patch signature generation, BINXRAY first computes the differences between the binary code of a VF and its corresponding PF, then creates a patch signature, which will be used in patch presence identification. It consists of three sub-components: binary instruction normalization, basic block mapping and valid trace generation.

**3.2.1 Binary Instruction Normalization.** The source code of a program is compiled into binary instructions by compilers. Due to compilation optimization, register allocation and assignment, address shifting, and other compilation settings, the binary instructions of two similar or same functions may be dissimilar after compilation. Since BINXRAY generates patch signatures by leveraging the differences between vulnerable and patched functions, it is important to eliminate changes that are introduced by compilers. Otherwise, the generated patch signatures will contain many irrelevant changes. Therefore, normalization is used to reduce the compiler introduced changes.

BINXRAY applies three normalization rules to binary instructions: **address normalization**: replacing concrete addresses by a symbolic term "address"; **memory normalization**: replacing indirect memory access by a symbolic term "mem"; and **register normalization**: replacing concrete registers by a symbolic term "reg". The normalization process is exemplified as follows:

#### Address normalization:

call 0x80488094 ->call address

#### Memory normalization:

mov [ebp], edx ->mov mem, edx

#### Register normalization:

mov ebp, esp ->mov, reg, reg

It is important to mention that BINXRAY does not normalize constants since some patches may change constants only in order to fix vulnerabilities, e.g., increasing the buffer size to fix a buffer overflow vulnerability. Normalizing constants will prevent BINXRAY from capturing this kind of patches.

**3.2.2 Basic Block Mapping.** Basic block mapping tries to map the same basic blocks between two functions. It is a technique to compute differences of the functions, in which unmatched blocks are regarded as the differences. There are basic blocks mapping algorithms in literature such as Bindiff [21] and Diaphora [5]. However, they sacrifice the accuracy in order to maximize the mapping speed and improve the scalability. Indeed, they use the hash values of the basic blocks to perform the mapping. If there are several duplicated blocks in one function that have the same hash value, they may fail to match the block with the correct one. For example, Figure 3(a) shows two CFGs need to be matched, as block1 and block1' are the same, block2, block3 and block3' are the same, and block2' is

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

	B1	B2	B3	B4
Ba	0.1	0.3	0.6	0.9
Bb	0.0	0.8	1.0	0.3
Bc	0.3	0.7	0.1	0.6

Figure 4: Example for Greedy Matching Algorithm

changed from block2 and different compared to others. Existing approaches can match block1 with block1'. However they may mismatch block2 with block3' since they share the same hash value as shown in Figure 3(b). The inaccuracy mapping will significantly affect the validity of the generated patch signatures. Therefore, we propose a new basic block mapping algorithm, which takes the syntax and context information of the basic blocks to alleviate the hash collision problem (i.e., different basic blocks have same hash value). Our method can complete mapping like Figure 3(c), where block2 and block2' can be recognized as CBBs correctly. We manually verified 36 real cases, that duplicate blocks turn up in one function. Our experimental results show that it outperforms the existing mapping algorithms in [5, 21].

Our basic block mapping algorithm first computes the hash values of basic blocks in both functions, based on their normalized instructions. Then it puts all the basic blocks with same hash value into the same basket. After that, if a basket has exactly two basic blocks from two different functions respectively, these two basic blocks are matched. If a basket only have basic blocks from one function, then these blocks must be changed basic blocks (CBB). If a basket has multiple blocks from both functions, then it is nontrivial to connect a mapping between these basic blocks. To solve this problem, we leverage their structural information and propose a greedy algorithm to establish the mapping between two basic blocks using similarity scores. For each pair of basic blocks from two different functions in one basket, BINXRAY computes a similarity score between them using their structural information (note that their syntactic information is already encoded as hash values). The similarity score of two basic blocks is computed by calculating the edit distance of the normalized instruction sequence in their adjacent basic blocks. If a basic block has more than one adjacent basic blocks, the similarity score will be weighted according to the control flow. The aggregated weight of the in-degree is normalized to be the same as the out-degree. After obtaining the pair-wise similarity scores of basic blocks, all the pairs will be put in a matrix. BINXRAY iteratively selects a pair of basic blocks from the matrix that has the highest similarity score, regards it as a pair of matched basic blocks and removes them from the matrix, until no more pair of basic blocks can match. Finally, all the remaining basic blocks in the matrix are regarded as CBBs. After computing the basic block mapping in one basket, the information will be propagated to other baskets as the contexts for other basic blocks.

**Example.** Suppose there are seven basic blocks Ba, Bb, Bc, B1, B2, B3, and B4 in one basket, where Ba, Bb, and Bc are from a VF, and the others are from its corresponding PF. The similarity scores of basic block pairs are computed as shown in Figure 4(a). The greedy algorithm first selects the pair (Bb,B3) which has highest similarity score 1.0 and removes them from the matrix, resulting the matrix as shown in Figure 4(b). Repeating this procedure, the pairs (Ba, B4) and (Bc, B2) are matched, the resulting matrixes are shown in Figure 4(c) and Figure 4(d). Finally, the remaining basic block is B1 from the PF which is regarded as a CBB.

**3.2.3 Valid Trace Generation.** To precisely express the patch signature of the given VF and PF, BINXRAY generates two sets of valid traces, one for VF and the another for PF. Based on the basic block mapping results between the VF and PF, it locates all the CBBs of these two functions. BINXRAY identifies the boundary basic blocks (BBBs) for each CBB, where a BBB is either an adjacent basic block of the CBB but not a CBB, or the root or leaf of the CFG of the function. A valid trace (VT) is a sequence of consecutive basic blocks that starts and ends with some BBBs, crosses at least one CBB, without any loops. If a loop occurs, we will flat (to treat the loop as being iterated once) it so that they will not affect the number of traces. Since BINXRAY relies mostly on syntax information, flatten the loop is a good choice which does not alter the syntax much. To build the VT, we put all the CBBs and BBBs into one connected graph. All the BBBs are the root and leaf nodes of the graph; and the CBBs are internal nodes. The valid traces are all the possible paths in the graph. BINXRAY generates two sets of VTs for the VF and PF, which are regarded as the patch signature. We denote by  $T_1$  the set of VTs of the VF, and  $T_2$  the set of VTs of the PF.

**Example.** Recalling the running example, CBBs in the function `dtls1_process_heartbeat()` in the version 1.0.1f (cf. Figure 1(a)) are: A, C, D, and F, and the BBBs in this function are: B, E, K, G. The CBBs in the patched function (cf. Figure 1(b)) are: A', C', D', E', F', G', I', L', M' and N', and the BBBs in this function are: B', H', P', J' and U'.

The VTs of the function `dtls1_process_heartbeat()` in version 1.0.1f (i.e., Figure 1(a)) are:

$$\begin{array}{ll} A \rightarrow B; & B \rightarrow C \rightarrow D \rightarrow E; \\ A \rightarrow C \rightarrow D \rightarrow E; & B \rightarrow C \rightarrow D \rightarrow G; \\ A \rightarrow C \rightarrow D \rightarrow G; & B \rightarrow C \rightarrow K; \\ A \rightarrow C \rightarrow K; & E \rightarrow F \rightarrow G; \end{array}$$

Similarly, the VTs for the patched function (i.e., Figure 1(b)) are:

$$\begin{array}{ll} A' \rightarrow B'; & A' \rightarrow C' \rightarrow L' \rightarrow U'; \\ A' \rightarrow C' \rightarrow D' \rightarrow M' \rightarrow U'; & A' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow N' \rightarrow U'; \\ A' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow G' \rightarrow H'; & A' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow G' \rightarrow J'; \\ A' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow P'; & B' \rightarrow C' \rightarrow L' \rightarrow U'; \\ B' \rightarrow C' \rightarrow D' \rightarrow M' \rightarrow U'; & B' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow N' \rightarrow U'; \\ B' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow G' \rightarrow H'; & B' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow F' \rightarrow G' \rightarrow J'; \\ B' \rightarrow C' \rightarrow D' \rightarrow E' \rightarrow P'; H' \rightarrow I' \rightarrow J'; & \end{array}$$

The VT sets can be optimized by merging the VTs which are connected end to end. The advantage of combining CBBs with BBBs to form the VTs is twofold. First, it pinpoints changes induced by vulnerability patching. Figure 5 has shown the real-world vulnerability CVE-2015-1790 in OpenSSL as an example. The CFG on the left part shows the related function `PKCS7_dataDecode()` in OpenSSL 1.0.1 l. The right part presents the detail of the function

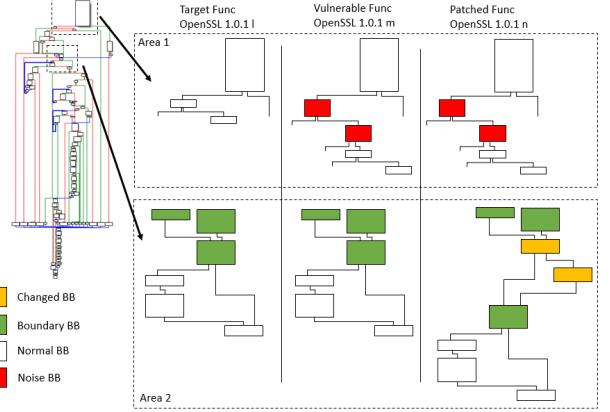


Figure 5: Justification of Using Boundary Basic Blocks

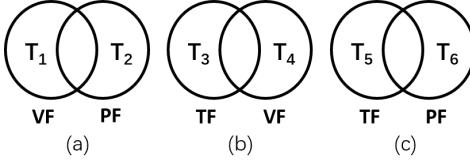
in three consecutive versions. In OpenSSL 1.0.1 l, the function is vulnerable, and we regard it as a target. In 1.0.1 m, the function has been updated (the red color blocks) in Area 1, but it is still vulnerable. In 1.0.1 n, the vulnerability has been fixed by adding sanity checks (the yellow blocks) in Area 2. BBBs help to locate the areas which are related to the patch signatures in target functions. Changes outside the areas will be filtered out. If we use the signature to detect the patch presence in the target function (1.0.1 l), according to the BBBs, only changes in Area 2 will be considered. Other changes will be treated as noises so that they will not affect the patch presence identification. On this example, using BBBs (green blocks), BINXRAY can focus on the changed (yellow) blocks, while neglecting the noises (red blocks).

Second, it significantly reduces the number of basic blocks and traces used in signature generation and patch presence identification. As shown in Figure 5, the function `PKCS7_dataDecode()` has around 100 basic blocks, which can form thousands of different traces. If we enumerate all the blocks and traces to build the signature, the signature size would be too large and the patch identification process would take a long time. Using BBBs, the signature size is reduced to 5 basic blocks and 2 traces. Our experimental results show that using BBBs significantly improves BINXRAY's performance. Indeed, BINXRAY can finish analysis in less than one second even on a large function, while it will takes more than ten minutes, if the entire function is used to build the signature.

### 3.3 Patch Presence Identification

Patch presence identification predicates whether a TF has been patched or not. The key idea is check whether the TF is more similar to VF or PF. If the TF is more similar to VF than PF, then the TF is regarded as a vulnerable function, otherwise it is regarded as a patched one. The patch presence identification consists four parts: trace generation, trace reduction, similarity comparison and decision algorithm.

**3.3.1 Trace Generation.** To match patch signatures in target function, BINXRAY first creates four sets of valid traces from VF, PF, and TF:  $T_3$ ,  $T_4$ ,  $T_5$  and  $T_6$ , as shown in Figure 6, where  $(T_1, T_2)$  is the patch signature.



**Figure 6: The Relationship between Valid Traces Sets**

To generate  $T_3$  and  $T_4$ , BINXRAY first builds the mapping between the VF and TF by using the basic block mapping algorithm in Section 3.2.2, and then extracts the CBBs of the VF and TF. Unlike the patch signature generation, BINXRAY reuses the BBBs generated between the VF and PF. It creates set  $T_3$  by combining the CBBs in TF with BBBs from patch signatures which are adjacent to those CBBs. One or more local CFGs can be constructed, and trace sets are generated by traversing those local CFGs. Similarly, it creates set  $T_4$  by combining the CBBs in VF with the corresponding BBBs. The reason for using BBBs is that it ensures only the relevant blocks are included in the sets. If there are some CBBs in other part of the function which are not related to the vulnerability, they will not form valid traces since no BBBs are adjacent to them. Therefore, it helps to remove the noises. Finally, the same procedure is applied to get the set  $T_5$  from the TF and the set  $T_6$  from the PF by leveraging the mapping between the TF and PF.

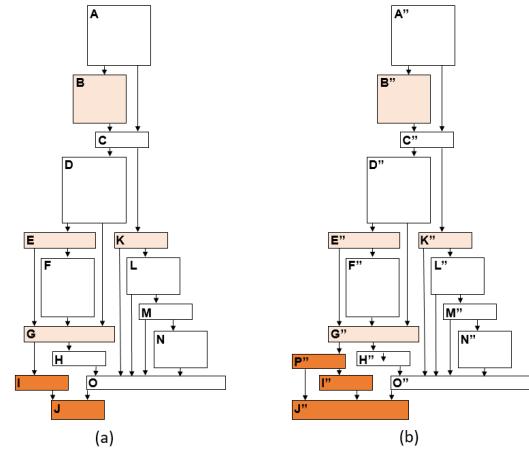
**3.3.2 Trace Reduction.** To further improve the prediction accuracy and performance, BINXRAY will eliminate irrelevant traces in the aforementioned trace sets ( $T_4$  and  $T_6$ ). For each valid trace  $t$  in the set  $T_4$ , if there does not exist any valid trace in  $T_1$  that contains the same CBB of  $t$ , the valid trace  $t$  will be eliminated from the set  $T_4$ , resulting in a reduced set of valid traces  $T_{41}$ . Similarly, BINXRAY will eliminate the irrelevant trace from  $T_6$ , by comparing with the traces in  $T_2$ , resulting in a reduced set of valid traces  $T_{62}$ . The reduction ensures that the remaining traces in  $T_{41}$  and  $T_{62}$  will contain some CBBs in the patch signature, which are considered as relevant in the patch presence identification.

**Example.** Figure 7 provides the complementary part for the running example in Figure 1. The VF in Figure 7(a) is the same as the one in Figure 1(a). Figure 7(b) shows the function in a modified version as the TF. The basic block  $P''$  is added into the TF, resulting in blocks  $(I, J)$  being marked as CBB in the VF in Figure 7(a). Recall that the BBBs  $(B, E, K, G)$  are in the patch signature, BINXRAY generates the trace set  $T_4$  for the VF, which contains only one valid trace as shown below:

$G \rightarrow I \rightarrow J;$

In trace reduction, each trace in the reduced sets needs to contain at least one block in the CBB sets of the patch signature. In Figure 1(a), the CBBs are:  $A, C, D$  and  $F$ . Therefore, the trace  $G \rightarrow I \rightarrow J$  will be removed by the trace reduction. By removing it, BINXRAY filters out the irrelevant traces and the final accuracy will be improved.

**3.3.3 Similarity Comparison.** BINXRAY performs patch presence identification by calculating the similarity score between trace sets and patch signatures. To compute similarity score for each pair of trace sets, we first show how to compute similarity score for a pair



**Figure 7: Running Example Continues: function `dtls1_process_heartbeat()` in OpenSSL**

of traces. For each trace in a trace pair, BINXRAY will first connect the basic blocks in the trace to form a sequential instruction trace by removing all the jump instructions. Then, two instruction traces are compared to obtain the similarity score. The score is computed according to the Equation (1).

$$\text{Sim}(t_1, t_2) = \frac{\max(\text{len}(t_1), \text{len}(t_2)) - \text{edit}(t_1, t_2)}{\max(\text{len}(t_1), \text{len}(t_2))} \quad (1)$$

BINXRAY computes the Levenshtein distance between two instruction traces, denoted as  $\text{edit}(t_1, t_2)$ . Then the distance is deducted from the maximum length of the traces and divided by the maximum length. The resulting score measures the normalized similarity between two traces.

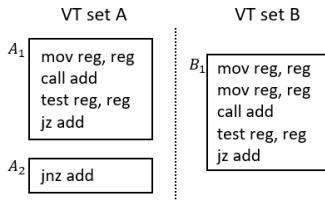
After having individual score of each pair of traces, the final similarity between two trace sets is computed according to the following equation.

$$\text{Sim}(T_1, T_2) = \sum_{\substack{t_1 \in T_1 \\ t_2 \in T_2}} \frac{\text{Sim}(t_1, t_2) * (\text{len}(t_1) + \text{len}(t_2))}{|T_1| * \text{len}(T_2) + |T_2| * \text{len}(T_1)} \quad (2)$$

The similarity score of two trace sets  $\text{Sim}(T_1, T_2)$  is normalized according to the trace length. The score is timed by the length of the  $t_1$  and  $t_2$ , and divided by a base, which is the number of traces in the  $T_1$  times the total length of all traces in  $T_2$  add the number of traces in the  $T_2$  times the total length of all traces in  $T_1$ . It ensures that the final score is scaled within the range [0,1].

The equation guarantees that the longer traces contribute more weight than the shorter ones. The long traces are usually the ones that contain the patch information. Therefore, they should be more contributive to the similarity score. The short traces usually come from the small and isolated blocks in the function, which might be the noise with little patch information. They should have less weight so that the overall score will not be affected by them.

**Example.** Figure 8 displays two sets of valid traces for the similarity computation. BINXRAY will compare every trace in A with every trace in B. According to Equation (1), the similarity score of (A1, B1) is 0.8 (4 out of 5 instructions match) and the similarity score



**Figure 8: Trace Sets Example for Similarity Score Calculation**

of  $(A_2, B_1)$  is 0 (none of the instructions match). Then, according to Equation (2) the overall similarity is normalized based on the trace length, which is 0.48 ( $= 0.8 * 9/15 + 0 * 6/15$ ). Due to the normalization, although the trace  $A_2$  is different from  $B_1$ , it has less effect to the overall similarity score than trace pair  $(A_1, B_1)$ . It helps to mitigate the noise by lowering the weight of the short instruction traces.

**3.3.4 Decision Algorithm.** We propose a decision algorithm to determine whether a target function has been patched. The core idea of this algorithm is to infer the relationship of functions by leveraging their differences and similarities. It is more accurate in dealing with partial similarity problem. Figure 6 shows the relationship between valid traces sets. Each trace set represents the unique part related to vulnerability in the corresponding function. There are three cases described below:

**CASE 1:  $T_1$  and  $T_2$  are both non-empty.** If the TF has been patched, then the difference between the TF and VF should be more significant than the one between the TF and PF. Otherwise if the TF is vulnerable, then the difference between the TF and PF should be more significant than the one between the TF and VF. Therefore, in this case, BINXRAY checks whether  $\text{Sim}(T_3, T_2) > \text{Sim}(T_5, T_1)$  holds or not. If  $\text{Sim}(T_3, T_2) > \text{Sim}(T_5, T_1)$  holds, then we say the TF has been patched; otherwise it is vulnerable.

**CASE 2:  $T_1$  is empty and  $T_2$  is non-empty.**  $T_1$  is empty meaning that the patch has added some fresh code. If the TF has been patched, then  $T_3$  will be similar to  $T_2$  and  $T_{62}$  will be empty. Otherwise if the TF is vulnerable, then  $T_{62}$  will be similar to  $T_2$ , and  $T_3$  will be empty. In this case, BINXRAY checks whether  $\text{Sim}(T_2, T_3) > \text{Sim}(T_2, T_{62})$  holds or not. If  $\text{Sim}(T_2, T_3) > \text{Sim}(T_2, T_{62})$  holds, we say the TF has been patched; otherwise it is vulnerable.

**CASE 3:  $T_2$  is empty and  $T_1$  is non-empty.**  $T_2$  is empty meaning that the patch deleted some code. Similar to CASE 2, we say the TF is patched if  $\text{Sim}(T_1, T_{41}) > \text{Sim}(T_1, T_5)$ ; otherwise, it is vulnerable.

## 4 EVALUATION

### 4.1 Experiment Setups

Our approach takes the binary code of pairs of vulnerable and patched functions as input. We choose to collect vulnerabilities from widely-used libraries with Common Vulnerabilities and Exposures (CVE) identifiers. There are websites providing CVE information, such as CVE Details [3], NVD [7] and CVE List [4]. In addition, some open source library websites also provide well-documented security updates, such as OpenSSL [9]. We extract CVE information from those websites, which consists of the CVE IDs, names of involved

**Table 2: Real-World Programs and Their Patch Presence Identification Results**

Program	CVE (#)	Version (#)	Function (#)	Function Accuracy	CVE Accuracy
Openssl	70	22	2280	95.04%	95.72%
FFmpeg	52	55	1486	89.64%	92.37%
Libxml2	37	12	852	94.95%	97.92%
Freetype	57	19	616	93.34%	98.14%
binutils	147	10	412	90.33%	96.24%
Tcpdump	88	3	273	100%	100%
Libpng	4	18	162	95.68%	100%
Openvpn	6	7	53	100%	100%
Sqlite3	6	10	53	81.13%	93.75%
Libeifx	7	12	28	75%	100%
Libxslt	3	5	15	100%	100%
Expat	2	4	8	87.50%	100%
Total	479	-	6238	93.31%	96.87%

functions, versions of programs that are vulnerable or patched. For each CVE used in the experiments, we manually confirmed that its information is valid. In future, we plan to collect more vulnerabilities from security update commits using security-related commit identification approaches, such as the approach in [42] for source code and [47] for binary executable.

Using the collected CVE information, we download the source code of target programs, compile them into binary executables using gcc with default optimization (-O2), and then dump the binary code of the functions using the binary disassembler IDA Pro [6]. For each function with a CVE, we save two versions: its vulnerable and patched versions respectively. For each CVE the changed functions before the patch commit are considered as vulnerable and functions after the patch commit are considered as patched.

In the experiments, we aim to answer the following research questions:

**RQ1** - How accurate is BINXRAY for patch detection?

**RQ2** - What is the (breakdown) performance of BINXRAY?

**RQ3** - How is the result of BINXRAY, compared to other related works?

**RQ4** - What are the useful applications of BINXRAY?

**RQ1** and **RQ2** evaluate whether BINXRAY meets **P1** and **P2** described in Section 1. As BINXRAY is designed not to use any source code level information, it meets **P3** by nature.

In the experiments, BINXRAY utilizes IDA Pro [6] to disassemble binaries and dump the binary function information. BINXRAY is implemented in Python 2.7 with more than 2K lines of code, and supports for Intel X86 32bit and 64bit, ARM 64bit architectures. All the programs run at HP desktop Z640 with CPU E5-2697 at 2.60GHz and 64GB memory. We have released all the experimental data at our website [8].

### 4.2 Accuracy Evaluation (RQ1)

In this evaluation, we aim to test whether BINXRAY can successfully predict the patch presence given a potentially vulnerable function. To prepare the data for the experiments, we have selected 12 different real-world programs across different domains such as cryptography, database, and image processing. The selected benchmark

programs are diverse, representative, and widely used in the literature [15, 17, 31, 48] and have adequate amount of well-documented vulnerabilities with CVE numbers. For each program, we collected the binary level vulnerable and patched functions by using function matching with its CVE function signatures. We manually confirmed that all the functions used in the experiments are either vulnerable or patched so that we can evaluate the patch identification accuracy in controlled settings. Then, we collected binaries in all the versions for each program as targets. Note that in some versions of binaries, there are a few vulnerable functions that are inlined or removed from the binary during the compilation process. We exclude these cases in our experiments. In total, we collected 479 CVEs of these programs, and collected all the functions affected by these CVEs. After matching those functions in different versions of binaries, we obtained 6238 target functions as test data. We run the patch identification process of BINXRAY on these functions and compared its results with the ground truth to obtain the accuracy measurement.

Table 2 shows the prediction results on these functions. The first four columns report the program name, the number of CVE collected, the number of versions tested in the experiments, and the number of functions to be predicted respectively. The fifth column provides the accuracy in predicting the patch presence for each of the target functions found by BINXRAY. The last column provides the accuracy in predicting whether the CVE (vulnerability) has been patched or not. When a CVE affects only one function, BINXRAY predicts the CVE still exists in the binary if this function is predicted as vulnerable, and predicts the CVE has been patched if this function is predicted as patched. When one CVE contains multiple vulnerable functions, BINXRAY will predict as vulnerable, if more than one of the functions are predicted as vulnerable. Otherwise, it is predicted as been patched.

The overall accuracy is above 93% for function level patch identification and over 96% for predicting the patches of CVEs. The results show that for most of the programs with a few CVEs (i.e., TCPdump, Libpng, Openvpn, Libeifx, Libxslt, and Expat), BINXRAY can identify the patch presence of the CVE with 100% accuracy. The reason for the high accuracy is twofold. First, some of the functions are very stable with only a few changes across all the versions. Therefore, the patch signature will be the dominant change that contributes the most to the final result, and make the function easy to be identified correctly. Second, the functions that contain the CVEs are usually large, so that different changes in one function will have a high chance to be placed in different locations. For multiple changes in one function, since BINXRAY is good at handling the cases where the patch-related changes and other changes are separated, it can accurately identify the patches and correctly predict its presence.

**False Prediction Discussion:** We have manually examined the false prediction cases made by BINXRAY. Most of them are caused by the multiple times of changes made at the same location of the function. Since most of the original code in patch related area is modified by multiple changes, the patch signature generated from the old version cannot match the newly modified functions. Therefore, BINXRAY will make false predictions.

There is a special false prediction case in the experiment of OpenSSL. CVE-2015-1791 [2] is a race condition vulnerability. It

**Table 3: Performance of BINXRAY**

Program	Basic Blocks (#)	Total Time (s)	Time per Function (ms)
OpenSSL	54.08	561.95	246.47
FFmpeg	90.18	6.42	4.32
Libxml2	60.47	55.61	65.27
Freetype	66.99	271.37	440.53
Binutils	83.21	375.16	911.32
TCPdump	45.31	28.57	104.65
Libpng	26.40	12.99	80.20
Openvpn	20.20	2.91	54.90
Sqlite3	291.16	449.27	8476.85
Libeifx	208.15	82.87	2959.66
Libxslt	95.07	0.22	14.67
Expat	15.38	0.09	11.25
Average	88.05	-	296.17

is patched in function `ss13_get_new_session_ticket()` at version 1.0.1n. BINXRAY can successfully distinguish the patch and unpatched functions from version 1.0.1a to 1.0.1n. However, after 3 versions at 1.0.1q, the patch change has been reverted back into the original functions. Therefore, BINXRAY classifies the version after 1.0.1q as vulnerable. In fact, the vulnerability has been fixed in another function. Even for human experts, it will take significant efforts to understand that the vulnerability patch has been replaced by other changes in different places. Theoretically, BINXRAY has made a correct prediction that the TF is vulnerable. However, due to the patches in other functions, the vulnerability is patched, resulting in a false positive in CVE prediction.

**Answering RQ1:** The results on the real-world programs show that BINXRAY can effectively identify the patch in the target functions. It has an average 93.31% accuracy in predicting the patch presence and 96.87% accuracy in identifying CVEs.

### 4.3 Performance Evaluation (RQ2)

Table 3 reports the performance to complete the patch presence identification for the 12 real-world programs. The second column reports the average number of basic blocks in the target function, the third column gives the total overhead for all functions, and the last column reports the average time overhead to identify patch for one function. According to the table, the number of basic blocks in each function is 88.05 on average. BINXRAY is very efficient, which can make prediction in 296.17ms per function on average.

In the experiment, running BINXRAY on Sqlite3 took the longest time (8476.85 ms per function). We manually verified the program and located one function that caused the problem. Specifically, there is a very large function, named `sqlite3VdbeExec()`, with more than 1000 basic blocks. The trace sets generated by BINXRAY consists of thousands of traces, which resulted in the significant time consumption. The predictions on the other functions finished within reasonable time period.

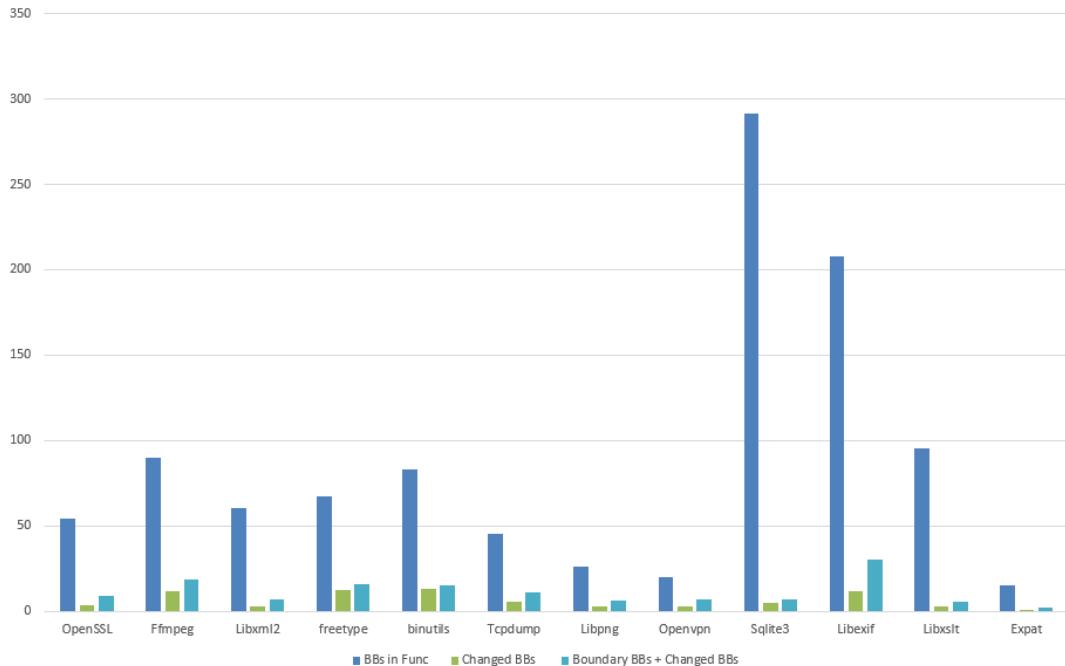


Figure 9: Basic Blocks in Original Functions vs Basic Blocks Used in BINXRAY

Table 4: Comparison with Binary Function Matching

Tool	Vul (#) Func.	Patched Func.(#)	FP	FN	Time per func.
BINXRAY	1412	868	29 (1.27%)	84 (3.68%)	246.47ms
BinGO-E			868 (38.07%)	0 (0%)	428.09ms

Figure 9 reports the number of basic blocks that were used by BINXRAY in the experiment for the 12 real-world projects (a larger version of the figure can be found at [8]). For each project, the first bar stands for the average number of basic blocks inside the target functions. The second bar shows the number of CBBs in the function, and the third bar shows the basic blocks that used by BINXRAY for signature generation (CBBs and BBBs). From these bars, we can observe that BINXRAY effectively reduces the size of the basic block sets by 82.55% on average to perform the signature generation and patch identification. The number of basic blocks in the changed area used in BINXRAY is much smaller than the number of blocks in the entire function. The reduction of basic blocks significantly improves the speed and accuracy of patch identification and make BINXRAY practical for large real-world programs.

**Answering RQ2:** BINXRAY effectively reduces the number of basic blocks used by 82.55%. BINXRAY can perform vulnerability matching with very low overhead so that it is scalable for real-world binary programs.

#### 4.4 Related Works Comparison (RQ3)

**Comparison to Function Matching Works.** To compare with the existing binary function matching techniques for vulnerability matching, we have selected BinGo-E [48] from recent state-of-the-art works as baseline tool. BinGo-E is selected because it has the best performance and its source code is successfully requested from the authors. We use the Openssl data set (2280 target functions of 70 CVEs in 22 versions) to test BinGo-E. Table 4 shows the results of BinGo-E against BINXRAY. The second and third columns report the number of vulnerable functions and patched functions in the ground truth. The fourth and fifth columns provide the false positive and false negative. The last column reports the time used to process one function. BinGo-E is an accurate function matching tool, which can match all the functions in data set. However, since it cannot determine the patch presence, it labels all the matched function as vulnerable. However, there are 868 functions having been patched, which results in a high false positive rate (38.07%). In comparison, BINXRAY has a much lower false positive rate with a reasonable low false negative rate. BINXRAY takes half of the time to make prediction on one function.

**Comparison to other Patch Identification Works.** In this experiment, we compare BINXRAY with FIBER [49], a state-of-the-art patch presence identification tool for Android. We have obtained the patch data set from the author of [49]. It contains 107 vulnerabilities in different versions of Android kernels. We managed to select an Android kernel, and construct two versions of it, one reproduces all of the CVEs and the other patched all of them. Excluding the mismatching between the kernel and patch and the impact of compilation, eventually, we manually verify 76 vulnerabilities in the

**Table 5: Comparison with Patch Identification**

Tool	Data	Accuracy	Time (per Vul.)
FIBER	107 CVEs	94% aver.	min: 12.59s; max: 23.74s
BINXRAY	76 out of 107	97.37%	49.06ms

compiled Android binaries (Samsung bullhead configuration) and test the performance of BINXRAY on them.

Table 5 shows the result of FIBER and BINXRAY on detecting the Android vulnerability patches. The accuracy of FIBER is calculated based on the 107 CVEs as stated in its paper, while BinXray is based on the 76 manually verified CVEs. Overall, BINXRAY has a slightly better performance than FIBER in terms of accuracy. It is much faster than FIBER, since FIBER requires heavy program analysis. Moreover, FIBER needs the source code information so that it is not applicable for closed source software. BINXRAY can work in larger scope, since it only requires the binary diffs.

**Answering RQ3:** BINXRAY outperforms the vulnerability matching tool BinGo-E. Compared with patch identification tool FIBER, BINXRAY achieves higher accuracy and speed without using the source code information.

## 4.5 Applications (RQ4)

**4.5.1 Binary Program Version Identification.** One potential application of BINXRAY is to determine the version of unknown binary programs. Version detection is an important step in binary analysis, especially for software security [16]. In different versions of programs, there are changes to update or patch functions. Since BINXRAY can precisely detect these changes, it can leverage them as signatures to determine the program versions. We conducted experiments in identifying program versions with OpenSSL binaries. First, we used function diffing technique to obtain a changed functions list for OpenSSL. Table 6 has shown the OpenSSL version 1.0.1 a to 1.0.1 e with their corresponding changed functions. The original function is labeled as tag0. After one change has been made, the tag number will be increased by one (tag1). These binary functions are fed into BINXRAY to learn the signatures. After extracting the signature, BINXRAY can correctly identify the tags for given binary functions in the table. To determine the binary version, BINXRAY will predict all the 11 functions' tags and match them against the table. If all the 11 functions have tag0, the given binary should be version 1.0.1 a. If the functions `rc4_hmac_md5_cipher` and `ssl23_client_hello` are tag1 and the rest functions are tag0, the given binary should be version 1.0.1 b, etc. The experimental results show that, by identifying function tags, BINXRAY can precisely identifies the binary versions.

**4.5.2 IoT Device Vulnerability Detection.** Firmware in IoT devices utilize many open source libraries, which may contain software vulnerabilities. However, these libraries are often in binary format without source code information. Therefore, as an outsider, it is difficult to know what changes have been made to the libraries and how many vulnerabilities remain. In this situation, BINXRAY can be used to detect and validate the vulnerabilities in libraries that used by firmware. To demonstrate this capability, we have used BINXRAY

**Table 6: OpenSSL Version Timeline**

Function Name	OpenSSL 1.0.1				
	a	b	c	d	e
<code>rc4_hmac_md5_cipher</code>	tag0	tag1	tag1	tag1	tag1
<code>ssl23_client_hello</code>	tag0	tag1	tag1	tag1	tag1
<code>dtls1_enc</code>	tag0	tag0	tag1	tag2	tag2
<code>tls1_enc</code>	tag0	tag0	tag1	tag2	tag2
<code>cms_EncryptedContent...</code>	tag0	tag0	tag1	tag1	tag1
<code>ssl3_get_client_hello</code>	tag0	tag0	tag0	tag1	tag1
<code>www_body</code>	tag0	tag0	tag0	tag1	tag1
<code>do_ssl3_write</code>	tag0	tag0	tag0	tag1	tag1
<code>ssl3_cbc_digest_record</code>	tag0	tag0	tag0	tag0	tag1
<code>tls1_cbc_remove_padding</code>	tag0	tag0	tag0	tag0	tag1
<code>ssl3_cbc_copy_mac</code>	tag0	tag0	tag0	tag0	tag1

to search the vulnerabilities in 7 real-world IoT device firmware that contain Openssl library. We extracted and parsed the binary files in those firmware and used patch signatures generated from Openssl in previous experiments to detect vulnerabilities. In total, it successfully identifies that 49 vulnerabilities have been patched and 48 vulnerabilities still remain in the firmware. The results have been manually confirmed. Due to the complex composition of the firmware and the difficulty of parsing firmware, the average accuracy of BINXRAY in this experiment is 81.5%. More details can be found at [8]. The result shows that BINXRAY can provide an effective solution in the scenario mentioned above.

**Answering RQ4:** Experimental results have demonstrated the capability of BINXRAY in determining the exact version of the real-world binaries. Moreover, it can precisely detect CVEs in firmware.

## 4.6 Threats to Validity & Future Work

BINXRAY has some limitations, which need to be overcome. First, the prerequisite of our work is accurate function matching. If the function matching algorithm cannot find the target functions, BINXRAY cannot proceed to identify the patches. Therefore, if the target function has been significantly changed so that the function cannot be matched, our algorithm cannot work.

Second, the experiments are conducted on binaries compiled for Intel X86 32 bit, 64 bit, ARM 64bits system. BINXRAY is designed to support any kind of architectures as long as the signature is extracted within the same type of architecture. However, BINXRAY currently does not support cross-architecture patch detection. In the future, we plan to use the intermediate representation to lift the binary instruction to higher level to support the cross-architecture comparisons.

Third, as discussed in Section 4.2, BINXRAY is not able to handle the case where a function receives multiple changes at the same location in different versions. Such changes may mislead BINXRAY to make incorrect decisions. A possible solution is to use inter-function semantic analysis to find the root cause of the vulnerability and check whether the problem has been addressed or not. However, heavy program analysis will significantly decrease the performance.

Therefore, we need to design algorithms to make trade-off between the performance and accuracy.

## 5 RELATED WORK

In this section, we review the most closely related works in two directions: binary function matching and binary patch identification.

**Binary Function Matching.** Binary-level function similarity matching has drawn much attention because of its important applications, e.g., copyright checking, malware analysis and binary program auditing [17, 19, 25, 40, 43]. Saebjornsen et al. [40] attempt to normalize the binary instructions and utilize the structural information to match the similar function codes. BinHunt [22] and iBin-Hunt [35] use symbolic execution and taint analysis to determine the differences between functions. The heavy program analysis methods introduce high overhead. Therefore, various works try to address the problem by introducing lightweight matching methods. TRACY [15] proposes to use  $k$ -tracelet to address basic block merging problem in matching binary functions. DiscovRE [18] tries to identify similar vulnerable functions across architectures with the help of numeric and structural features. BLEX [17] adopts seven dynamic features by executing the functions to be tolerant for small changes. BinGo [11] and BinGo-E [48] try to find similar function pairs by using multiple kinds of features (i.e., syntax, semantics, and emulation features).  $\alpha$ Diff [31] uses deep neural networks to automatically learn features from raw bytes of the binaries. Bourquin et al. [10] present a polynomial algorithm to improve the accuracy in calculating the function similarity. Xu et al. [46] use neural networks to embed the features and match the embedded graph to improve the matching speed and accuracy. Luo et al. [33] extract semantic features of the functions and match the functions with obfuscations. Genius [20] builds CFG of functions, embeds the CFG into high level numeric features and searches for known vulnerability in the firmware. Pewny et al. [38, 39] propose cross-architecture bug search by translating the binary instructions into intermediate representations. Lin et al. [30] search the bug in the firmware by using the attributed CFG with support vector machine technique. Hu et al. [23] try to rebuild the argument and indirect jumps in the binary to increase the matching precision. Hu et al. [24] combine the static and the dynamic approaches to detect the code clones with obfuscation. David et al. [13] use statistical probability to measure the similarity of the program strands. Ming et al. [36] determine whether two program execution traces are similar by using the system call sliced segment equivalence checking algorithm.

These works aim to find matching function pairs with the tolerance of small changes. Therefore, they may generate high false positive by matching the vulnerable functions with the patched functions. Our work tries to focus on summarizing the differences between the vulnerable functions and their patched versions. Thus, it can significantly improve the accuracy of similarity-based vulnerability detection by filtering out the patched functions.

**Source Code Function Matching and Patch Analysis.** There are also many works which try to search for similar code or similar functions at source code level (e.g., [26, 28]), while ours works at the binary code level. For known vulnerabilities, source-level patch identification can be done directly by checking whether the code has been updated with patches. Binary-level identification is more

challenging since functions will change due to factors (compiler type, version and settings, program updates, and architecture). One needs to distinguish the patch changes out of different changes to perform patch identification.

**Binary Patch Analysis and Identification.** Several studies [29, 42, 50] have surveyed the vulnerability patches in the real-world software to understand the common characterizations of patches. BugTrace [12] tries to connect the link between the vulnerabilities and their fixes via patch analysis. Soto et al. [41] study the patch behaviors in Java program. Kim and Notkin [27] leverage the understanding of patch diffs to help the programmer to find vulnerabilities. These works have shown the detailed studies for vulnerability patches in real-world software. However, they are on the source code level and are not designed specifically for identifying vulnerable functions.

Zhang and Qian [49] propose FIBER to predict whether the function in Linux kernel has been patched or not. It relies on source code information of the target function, so that the additional preprocessing is required for different projects. Our work uses only binary information so that it is more convenient and more suitable when the source code is not available. VMPBL [32] tries to further distinguish the patched and unpatched functions by building a vulnerable function database and a patched function database. Our work tries to automatically extract patch signatures for each CVE and does not require any prior knowledge. Spain [47] tries to identify the secretly patched vulnerabilities by comparing the semantic changes at binary level. It aims to verify whether a patch is security-related. Our work aims to identify whether the patch exists or not. Memlock [45] leverages memory usage to guide fuzzing to find vulnerabilities. Then, Wang et al. [44] propose to use memory layout recovering to find vulnerabilities and patches.

## 6 CONCLUSIONS

This work proposes BINXRAY to eliminate the false positives in vulnerable binary function matching by identifying the patches in the target functions without using source code level information. It can automatically extract the signatures of the vulnerability patches by diffing the vulnerable and patched functions. The patch signatures are used to match the target functions to determine whether they have been patched or not. The experimental results show that BINXRAY is able to achieve high accuracy with very low overhead. Moreover, BINXRAY can help to determine the binary level program versions and finds real-world vulnerabilities in IoT firmware.

## ACKNOWLEDGMENTS

This work was partially funded by National Key R&D Program of China (2018YFB0803501), the Fundamental Research Funds for the Central Universities, Singapore National Research Foundation, under its National Cybersecurity R&D Program (Grant Nos.: NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Grant Nos.: NRF2018NCR-NSOE003-0001), NRF Investigatorship (Grant Nos.: NRFI06-2020-0022), National Natural Science Foundation of China (NSFC) grants (No. 61772408, No. U1766215, No. U1736205, No. 61721002, No. 61632015, No. 61532019, No. 61761136011 and No. 61902306), and Alibaba Group through Alibaba Innovative Research (AIR) Program.

## REFERENCES

- [1] 2014. CVE-2014-0160. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.
- [2] 2015. CVE-2015-1791. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-1791>.
- [3] 2020. CVE Details. <https://www.cvedetails.com/>.
- [4] 2020. CVE List. <https://cve.mitre.org/index.html>.
- [5] 2020. Diaphora. <https://github.com/joxeankoret/diaphora>.
- [6] 2020. IDA Pro. <https://www.hex-rays.com/products/ida/>.
- [7] 2020. NVD. <https://nvd.nist.gov/>.
- [8] 2020. Open Source Data and Results for the Paper. <https://sites.google.com/view/submission-for-issta-2020>.
- [9] 2020. OpenSSL Vulnerabilities. <https://www.openssl.org/news/vulnerabilities.html>.
- [10] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 4.
- [11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.
- [12] Christopher S Corley, Nicholas A Kraft, Letha H Etzkorn, and Stacy K Lukins. 2011. Recovering traceability links between source code and fixed bugs via patch analysis. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 31–37.
- [13] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.
- [14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 392–404.
- [15] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *AcM Sigplan Notices* 49, 6 (2014), 349–360.
- [16] Ruian Duan, Ashish Bajlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. ACM, 2169–2185.
- [17] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.
- [18] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.
- [19] Mohammad Reza Farhadji, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Proceedings of the 8th International Conference on Software Security and Reliability*. 78–87.
- [20] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.
- [21] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment*, number P-46 in *Lecture Notes in Informatics*. Citeseer, 161–174.
- [22] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.
- [23] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 88–98.
- [24] Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. BinMatch: A Semantics-based Hybrid Approach on Binary Code Clone Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 104–114.
- [25] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. 309–320.
- [26] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.
- [27] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 309–319.
- [28] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: a scalable approach for vulnerable code clone discovery. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 595–614.
- [29] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2201–2215.
- [30] Hong Lin, Dongdong Zhao, Linjun Ran, Mushuai Han, Jing Tian, Jianwen Xiang, Xian Ma, and Yingshou Zhong. 2017. CVSSA: Cross-Architecture Vulnerability Search in Firmware Based on Support Vector Machine and Attributed Control Flow Graph. In *Dependable Systems and Their Applications (DSA), 2017 International Conference on*. IEEE, 35–41.
- [31] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018.  $\alpha$ Diff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 667–678.
- [32] Danjun Liu, Yao Li, Yong Tang, Baosheng Wang, and Wei Xie. 2018. VMPBL: Identifying Vulnerable Functions Based on Machine Learning Combining Patched Information and Binary Comparison Technique by LCS. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 800–807.
- [33] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.
- [34] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 24–35.
- [35] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*. Springer, 92–109.
- [36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*.
- [37] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hrithesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 180–190.
- [38] Jannik Peawy, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.
- [39] Jannik Peawy, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 406–415.
- [40] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 117–128.
- [41] Mauricio Soto, Ferdinand Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 512–515.
- [42] Yuan Tian, Julie Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 386–396.
- [43] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Eryue Zhuang, Ming Fan, and Zijiang Yang. 2017. Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Transactions on Software Engineering* 44, 5 (2017), 491–511.
- [44] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 718–728.
- [45] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE.
- [46] Xiaojuan Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 363–376.
- [47] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 462–472.
- [48] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* (2018).
- [49] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 887–902.
- [50] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 913–923.



# Identifying Java Calls in Native Code via Binary Scanning

George Fourtounis  
University of Athens  
Athens, Greece  
gfour@di.uoa.gr

Leonidas Triantafyllou  
University of Athens  
Athens, Greece  
leotriantafyllou@gmail.com

Yannis Smaragdakis  
University of Athens  
Athens, Greece  
smaragd@di.uoa.gr

## ABSTRACT

Current Java static analyzers, operating either on the source or bytecode level, exhibit unsoundness for programs that contain native code. We show that the Java Native Interface (JNI) specification, which is used by Java programs to interoperate with Java code, is principled enough to permit static reasoning about the effects of native code on program execution when it comes to call-backs. Our approach consists of disassembling native binaries, recovering static symbol information that corresponds to Java method signatures, and producing a model for statically exercising these native call-backs with appropriate mock objects.

The approach manages to recover virtually all Java calls in native code, for both Android and Java desktop applications—(a) achieving 100% native-to-application call-graph recall on large Android applications (Chrome, Instagram) and (b) capturing the full native call-back behavior of the XCorpus suite programs.

## CCS CONCEPTS

- Software and its engineering → Compilers;
- Theory of computation → Program analysis.

## KEYWORDS

static analysis, Java, native code, binary

### ACM Reference Format:

George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java Calls in Native Code via Binary Scanning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397368>

## 1 INTRODUCTION

Over two decades ago, Java ushered in the era of portable, architecture-independent application development. The attempt to make portable mainstream applications was originally met with skepticism and became a critical point in Java adoption debates, as well as in the focus of the language implementors. Within a few years, the Java portability story was firmly established, and since then it has been paramount in the dominance of Java—the top ecosystem in current software development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397368>

An often-overlooked fact, however, is that platform-specific (*native*) code is far from absent in the Java world. Advanced applications often complement their platform-independent, pure-Java functionality with specialized, platform-specific libraries. In Android, for instance, Almanee et al. [1] find that 540 of the 600 top free apps in the Google Play Store contain native libraries, at an average of 8 libraries per app! (The architectural near-monopoly of ARM in Android devices certainly does nothing to discourage the trend.) Desktop and enterprise Java applications seem to use native code much more sparingly, but native code still creeps in. Popular projects such as *log4j*, *lucene*, *aspectj*, or *tomcat* use native code for low-level resource access [11].

The presence of native code in a Java application hinders static analysis, at any level. Failing to analyze the native parts of an application causes analysis *unsoundness* [12, 30]. Concretely, Sui et al. recently showed that native code is a core threat in call-graph analysis [38]. Native code can call back to Java code, introducing false negatives in *reachability* analysis—the static analysis that finds which parts of the code are reachable. Reachability analysis is, for instance, critical for Android: as part of packaging an Android application for deployment, unreachable (*dead*) code is eliminated via automated analysis. Modern Android development depends on a manually-guided workflow (via the ProGuard [23] configuration language) to explicitly capture the Java entry points used by native code, so that reachable code does not get optimized away.

Other than such manual “fixes” of the analysis results, there are few solutions to the problems of native-code-induced analysis unsoundness. Reif et al. [33] find that “none of the [state-of-the-art Java analysis] frameworks support cross-language analyses”. In recent work, Lee proposes (as planned work) a hybrid Java/C static analysis [26] that addresses the issue. However, this heavyweight approach requires access and analysis of native source code, which is a severe burden in practice. Source code for third-party native libraries (and even metadata, such as DWARF information [8]) is typically unavailable to the Java developer. Furthermore, analyzing the source code of the native library is very hard—e.g., the code may be in any of several languages (C, C++, Rust, Go), many of which currently have no practically effective whole-program analysis infrastructure.

In this paper, we present a technique for finding the call-backs from native to Java bytecode,<sup>1</sup> via scanning of the binary libraries and cross-referencing the information with the Java code structure. Our approach recognizes uses of the Java Native Interface (JNI) API, which provides the bridge between native and Java code. Specifically, the technique identifies string constants that match Java

<sup>1</sup>Java bytecode may not necessarily be produced from Java source code. For simplicity, we merely write “Java code” in the rest of the paper, with the understanding that the applicability of the technique extends transparently to all languages producing Java bytecode.

method names and type signatures in native libraries, and follows their propagation (to find where method name strings are used together with type signature strings). In this way, the technique identifies entry points into Java code from native code, without fully tracking calls (i.e., call-graph edges) inside native code.

The resulting technique informs the static analyses of the Doop framework [5]. It is the first approach to effectively address unsoundness in static reachability analysis, in the presence of binary libraries. We evaluate the approach over large Android applications (Chrome, Instagram) and the native-code-containing programs in the XCorpus suite [11]. The two settings mandate different evaluation methodologies: for the Android applications, no native source code is available, yet the application has dynamic execution snapshots, showing Java methods called from native code. For the XCorpus programs, the bundled test suite does not exercise native callbacks, yet the native source is available for manual inspection. In both cases, our approach captures the full call-back behavior of the native code.

## 2 BACKGROUND

This section introduces the Java Native Interface specification (Section 2.1) and declarative static analysis (Section 2.2).

### 2.1 Java Native Interface

The Java Native Interface (JNI) [31] is an interface that enables native libraries written in other programming languages, such as C and C++, to communicate with the Java code of the application inside the Java Virtual Machine (JVM). The JNI is a principled form of a foreign function interface (FFI), a feature that mature programming languages usually incorporate as an escape hatch to third-party functionality or low-level operations. The JNI was first supported in JDK release 1.1, to improve the interplay of Java with native code (at a time when the JVM could itself be integrated with native code, especially Web browsers) [29].

The JNI allows programmers to use native code in their applications without requiring any change to the Java VM, which means that the native code can run inside any Java VM that offers JNI support. Via JNI, it is possible to create new Java objects and update them in native code functions, call Java methods of the same application from native code, and load classes and inspect them. This functionality is supported by an extensive API with appropriate methods and data structures that let native code interact with Java objects by using JVM concepts such as method and field descriptors. Such descriptors are full signatures for methods and fields, as they appear in bytecode, i.e. generics have been erased and types are represented by their low-level counterparts.

Figure 1 shows the “hello world” program in JNI, which exhibits the following features of the JNI API:

- The native function that implements native Java method `JNIEample.hello(Object arg)` is assumed to be named `Java_JNIEample_hello` and take a corresponding `jobject` argument.
- The native function also accepts a `JNIEnv` pointer for a reference to the JNI environment and a `jobject` for a reference to the receiver object (`this`). The `JNIEnv` argument points to a structure storing all JNI function pointers, which allow instantiation and use of

```
JNIEXPORT void JNICALL
Java_JNIEample_hello(JNIEnv *env,
 jobject thisObj, jobject arg) {
    printf("Hello World!\n");
    return;
}
```

Figure 1: “Hello world” native function example.

Table 1: Java Method Signatures Examples.

Method	Signature
void m1()	()V
int m2(long)	(J)I
void m3(String)	(Ljava/lang/String;)V
String m4(String, int[])	(Ljava/lang/String;[I)Ljava/lang/String;

objects, conversion between native strings and Java strings, and other functionality.

- The native function is decorated with macros that control the native code linking (`JNIEXPORT`) and call convention (`JNICALL`) for the specific platform for which the code will be compiled.

When using native code in an application, it is possible to call back Java methods from native functions. In order to call back a Java method, the programmer needs to find its method id object (of JNI type `jmethodID`). This object is looked up by giving the name of the containing class, the name of the method, and the low-level signature of the method (JVM method descriptor). The signature is a string of the form `(parameters)return-value` with some examples of methods and their signatures shown in Table 1.

The process of calling back a method starts by getting a reference to the object’s class by using method `FindClass()` [24]. Then, the method name and signature are given as arguments in the function `GetMethodID()` of the class reference and the method id is returned. The method id can be used to call the Java method using the right function for the specific case, such as `CallVoidMethod()`, `Call<Primitive-type>Method()` and `CallObjectMethod()`. As for the type of the returned value of the called method, this can be `void`, `<Primitive-type>` and `Object`, respectively. An example of the process for calling a Java method that takes an `Object` argument and returns an integer through native code is shown in Figure 2.

### 2.2 Points-To Analysis in Datalog

Datalog is a declarative logic-based programming language which is designed to be used as a query language for deductive databases. Our analysis uses the Doop framework, implemented in Datalog [5], which provides a rich set of points-to analyses (e.g., context insensitive, call-site sensitive, object sensitive) for Java bytecode. However, because of the modular way of context representation in the framework, code built upon any such analysis can be oblivious to the exact choice of context (which is specified at run-time).

Soot [42] is a framework that is used by Doop and is responsible for generating input facts for an analysis as a pre-processing step. By using this framework, Doop expects as input the bytecode form of a Java program, which means the original source is not

```
JNIEXPORT void JNICALL Java_JNIExample_callBack(JNIEnv *env, jobject thisObj, jobject obj) {
    jclass cls = (*env)->FindClass(env, "JNIExample");
    jmethodID method = (*env)->GetMethodID(env, cls, "exampleMethod", "(Ljava/lang/Object;)I");
    jint i = (*env)->CallIntMethod(env, thisObj, method, obj);
    printf("callBack(): i = %d\n", i);
}
```

**Figure 2: Call back Java method from native function example.**

needed but only the compiled classes are necessary. This allows for analyzing programs whose source code is not available. The set of asserted facts for a program is called its EDB (Extensional Database) in Datalog semantics. The relations that are generated and directly produced from the input Java program, and any relation data added to the asserted facts by user defined rules, constitute the EDB predicates.

```
VarPointsTo(obj, var) :-  
    AssignHeapAllocation(obj, var).  
VarPointsTo(obj, to) :-  
    Assign(to, from), VarPointsTo(obj, from).
```

**Figure 3: Simple Datalog example for IDB rules.**

Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation (Figure 3). The Datalog code of the example consists of two simple rules known as IDB (Intensional Database) rules in Datalog semantics. These two rules are used to establish new facts from a conjunction of facts that are already established. The rule of the first line constitutes the base case of the computation and states that upon the assignment of an allocated heap object to a variable, this variable may point to that heap object. The second rule is the recursive case which states that if the value of a variable is assigned to another variable, then the second variable may point to any heap object the first variable may point to. For instance, the recursive rule of line 2 states that if `Assign(to, from)` and `VarPointsTo(obj, from)` are both true for some values of `from`, `to`, and `obj`, then that `VarPointsTo(obj, to)` is also true.

### 3 HELLOJNI EXAMPLE

This section describes our technique informally using an easy example: a toy Java/C program that uses few string constants and is easy to disassemble. We will use standard command-line tools to show the essence of our technique, without yet introducing the additional modeling and filtering (which will come in Section 4).

Assume we have a Java program (Figure 4) that defines native functions (Figure 5).<sup>2</sup> Further, assume we compile this code on Linux, on x86-64 hardware.

A pure-Java static analysis of the resulting program will miss the calls from the native code for methods `newJNIObj()` and `callBack()`. However, we observe that necessary parts of the target methods (names and signatures) appear in the native code as constant strings.

<sup>2</sup>Code adapted from online JNI tutorial [24].

Investigating the problem, we first examine the resulting .so library, which is in ELF format. ELF (Executable and Linkable Format) [32] is a file format for binaries, libraries, and core files. In the ELF library, string constants reside in the .rodata section [27]. We use the `readelf` command [14] to view the ELF sections and find the address of section .rodata and then view the strings in .rodata (Figure 6). Since the section starts at address 2000, the strings “HelloJNI”, “[Ljava/lang/Object;Ljava/lang/Object;)I”, and “helloMethod” are at addresses 2035, 2050, and 2078 respectively.

Disassembling `Java_HelloJNI_callBack()` in Figure 7, shows `lea` instructions with a computed addresses in comments (computed by GDB<sup>3</sup>). These computed addresses are the references to the three strings found in the previous step. Thus, we can deduce that the native function uses these strings, one of which looks like a JVM signature. Also, these three strings match the type, name, and JVM signature of an existing Java method `HelloJNI.helloMethod()`, thus the native function may be calling this Java method.

Finally, we can map the native function back to the original Java method. While this in general can be arbitrarily difficult, in this example, we assume the default JNI behavior where `Java_HelloJNI_callBack()` will be linked to an existing native Java method `HelloJNI.callBack()`. Thus, we can form a call-graph edge from `HelloJNI.callBack()` to `HelloJNI.helloMethod()`.

The above steps assume that (a) the input program is easy to disassemble (debugging metadata or other information offers function boundaries), (b) there is a way to statically compute the references to the strings that are used, and (c) there is an easy way to map native functions back to their Java entry points. These assumptions may be violated: stripped or optimized binaries may be difficult to disassemble, address computation is platform- and compiler-dependent, and JNI linking can be complex. The next section presents the full technique, with more details on how to handle such difficult cases.

### 4 OUR TECHNIQUE

A summary of the steps of our technique follows:

- PRE A pre-processing step finds all method names and signatures in the Java code, forming a set  $\mathcal{M}_0$ .
- FILT The native code of each library  $n$  is scanned for strings that can be found in  $\mathcal{M}$ , resulting in set  $\mathcal{M}^n \subseteq \mathcal{M}_0$ .
- LOC The strings in  $\mathcal{M}^n$  are localized: for each native function  $f$  in library  $n$ ,  $\mathcal{L}_f^n$  is the set of strings being used in the body of  $f$ .
- INVO For each function  $f$ , any method whose name and signature can be found in  $\mathcal{L}_f^n$  is determined to be reachable from  $f$ , forming set  $\mathcal{I}_f^n$ .

<sup>3</sup><https://www.gnu.org/software/gdb/>

```

public class HelloJNI {
    static {
        System.load("libhello.so");
    }

    // Declare a native method sayHello() that receives nothing and returns void
    private native void sayHello();
    private native Object newJNIObj();
    private native void callBack(Object obj);

    static Object sObj;

    // Test Driver
    public static void main(String[] args) {
        HelloJNI hj = new HelloJNI();
        hj.sayHello(); // invoke the native method
        Object obj = hj.newJNIObj();
        System.out.println(obj.toString());
        sObj = hj.newJNIObj();
        System.out.println(sObj.toString());
        hj.callBack(new Object());
    }

    public int helloMethod(Object obj1, Object obj2) {
        System.out.println(obj1.hashCode());
        System.out.println(obj2.hashCode());
        return 1;
    }
}

```

Figure 4: Code of HelloJNI.java example file.

```

#include <jni.h>
#include <stdio.h>

// Implementation of native method sayHello() of HelloJNI class
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
    printf("Hello World!\n");
    return;
}

JNIEXPORT jobject JNICALL Java_HelloJNI_newJNIObj(JNIEnv *env, jobject thisObj) {
    jclass cls = (*env)->FindClass(env, "HelloJNI");
    jmethodID constructor = (*env)->GetMethodID(env, cls, "<init>", "()V");
    return (*env)->NewObject(env, cls, constructor);
}

JNIEXPORT void JNICALL Java_HelloJNI_callBack(JNIEnv *env, jobject obj) {
    jclass cls = (*env)->FindClass(env, "HelloJNI");
    jmethodID helloMethod = (*env)->GetMethodID(env, cls, "helloMethod", "(Ljava/lang/Object;Ljava/lang/Object;)I");
    jint i = (*env)->CallIntMethod(env, obj, helloMethod, obj, obj);
    printf("callBack(): i = %d\n", i);
}

```

Figure 5: Code of HelloJNI.c example file.

EDGE If the native function  $f$  implements a native Java method  $m$ , we form a call-graph edge from native method  $m$  to each method in  $\mathcal{I}_f^n$ .

#### 4.1 Step PRE

This step takes place during the pre-processing stage of the static analysis. In the Doop framework, this stage consists of “fact generation” (implemented using either Soot [42] or WALA [10]): the extraction of database tables with input program information, for later processing by Datalog rules. During the PRE phase, every

```
$ readelf --sections libhello.so
Section Headers:
[Nr] Name          Type      Address      Offset
     Size        EntSize   Flags Link Info Align
...
[13] .rodata       PROGBITS 0000000000002000 00002000
     0000000000001ab 0000000000000000 A 0 0 8
...
$ readelf -p .rodata libhello.so
String dump of section '.rodata':
...
[ 28] Hello World!
[ 35] HelloJNI
[ 3e] ()V
[ 42] <init>
[ 50] (Ljava/lang/Object;Ljava/lang/Object;)I
[ 78] helloMethod
...
```

**Figure 6: Viewing section .rodata in the example program.**

```
0x00000000000011d6 <+31>: lea    0xe58(%rip),%rsi      # 0x2035
0x00000000000011dd <+38>: mov    %rdx,%rdi
0x00000000000011e0 <+41>: callq *%rax

...
0x00000000000011fc <+69>: lea    0xe4d(%rip),%rcx      # 0x2050
0x0000000000001203 <+76>: lea    0xe6e(%rip),%rdx      # 0x2078
0x000000000000120a <+83>: callq *%rax
```

**Figure 7: Disassembled native function Java\_HelloJNI\_callback(), where string references are shown in comments.**

method encountered will save its name and JVM-level signature in set  $\mathcal{M}_0$ .

## 4.2 Step FILT

We find the native code of the application by reading all files recognized as dynamic libraries (e.g., with filenames ending in .so or .dll) from the input program. We extract the strings from the native code and only keep those that match actual method names or signatures in the program (by crossreferencing set  $\mathcal{M}_0$ , above). The resulting set is  $\mathcal{M}$ . As an optimization, if no method names or no signatures are found, we stop the analysis of this library (since the cross-product of names and signatures in the next step will be empty).

Finding strings is easy, as these are constants stored in special sections of the binary code (such as section “.rodata” in Linux ELF files). In practice, we can use the GNU “strings” utility, roll our own code to look for NULL-terminated strings, or use a special disassembler. We choose the last option and use Radare2,<sup>4</sup> a free and powerful reverse engineering framework, which is also employed in other steps of our approach. As a side effect, Radare2 gives us support for multiple hardware targets (x86, x86\_64, arm64, and armeabi) and operating systems (Linux, macOS, Windows).

This step also records global (or “static”) strings stored in the binary (such as the strings in the “.data” section in Linux ELF files). These strings are also useful for resolving JNI functionality related to dynamic linking and may be used later, in step CALL.

<sup>4</sup><https://rada.re/>

## 4.3 Steps LOC and INVO

At this point, we can already perform step INVO crudely, to match all found strings against the method names and type signatures (a.k.a. “descriptors”) of the Java code. If a method finds both its name and type signature in  $\mathcal{M}$ , then that method can be assumed to be called from the native code.

However, this can be too imprecise since a big native code library may contain many strings: strings used in different functions can accidentally match methods that will not be called in reality. Also, the resulting information is too basic: we can only determine the methods reachable from a native code library, which helps reachability computation but we cannot identify a call-site or other caller identity.

To address both above issues (imprecision and loss of invocation location), we perform step LOC, which finds which native code function uses which strings. Then, we can match name/signature strings per function, and solve the precision problem outlined above. Knowing the function can also help with the construction of a native call-graph edge  $N(f, m)$  from a native function  $f$  to a Java method  $m$ .

To find all places in the binary code where strings are referenced, we perform a “string cross-references” (a.k.a. “string x-refs”) analysis using Radare2. Radare2 can analyze binary code and convert it to a stack-based IR, which then can be analyzed and partially evaluated so that string references can be computed [9]. A string x-refs analysis is needed since string references may not appear

```
.decl PossibleNativeCodeTargetMethod(method:Method, function:symbol, file:symbol)

PossibleNativeCodeTargetMethod(method, function, file) :-
    NativeMethodTypeCandidate(file, function, descriptor),
    NativeNameCandidate(file, function, name),
    Method_SimpleName(method, name),
    Method_JVMDescriptor(method, descriptor).
```

**Figure 8: Datalog rule to find possible Java method calls by matching method names and type descriptors used in the same native function.**

```
// Mock arguments for methods called from native code.
MockValueConsMacro(mockId, frmType),
VarPointsTo(mockId, frm) :-
    PossibleNativeCodeTargetMethod(method, function, file),
    FormalParam(_, method, frm),
    Var_Type(frm, frmType),
    mockId = "<mock native object of type " + frmType + " from " + file + ":" + function + ">".

// Mock 'this' for methods called from native code.
MockValueConsMacro(mockId, type),
VarPointsTo(mockId, this) :-
    PossibleNativeCodeTargetMethod(method, function, file),
    ThisVar(method, this),
    Var_Type(this, type),
    mockId = "<mock receiver of type " + type + " from " + file + ":" + function + ">".
```

**Figure 9: Datalog rules to mock arguments and receivers of methods called from native code.**

as local constants but be formed at runtime, by Position Independent Code [27, Chapter 8], by some layers of obfuscation, or by the hardware architecture at hand.

We should note here that static disassembly is not a solved problem [2]. For example, binary files may contain complex constructs, such as overlapping code and inline data in executable regions. In our technique, function boundaries (and the assignment of symbols to specific functions where they are used) may not be always recovered with full accuracy.

The output of the LOC step is  $\mathcal{L}_f^n$ : the set of string constants of library  $n$  being used in the body of native function  $f$ . At the analysis level, this is expressed as relations NativeMethodTypeCandidate and NativeNameCandidate, which relate string constants that match method names and type descriptors (e.g., "(Ljava/lang/Object;Ljava/lang/Object;)I" in Figure 5) to a library filename and native function identifier. The final step is a Datalog query (shown in Figure 8) that matches method names and descriptors appearing in the same native function.

**4.3.1 Mock Objects.** Finding methods callable from native code does not immediately imply their full treatment in the static analysis. To fully statically model the effects of the native-to-Java call-back, we need to provide appropriate values to the call’s parameters, including the implicit “this” receiver passed to non-static methods. We create “mock” objects, i.e., appropriate artificial objects, for such arguments, following a policy of single-object-per-type-and-call. This is illustrated in the rules of Figure 9.

The first rule in Figure 9 is used to create a mock id for every argument and its type of every reachable method by joining the predicate PossibleNativeCodeTargetMethod(method, function, file)

and the predicates FormalParam(\_, method, frm) and Var\_Type(frm, frmType). The second rule has a similar behavior, creating a mock object for the receiver (“this”) of Java methods found in native code.

**4.3.2 Constructor Filtering.** Constructors are regular Java methods that are called when new objects are created. Native allocations are common in native code [38], so constructors are often called by native libraries. Since all constructors have the same low-level name (`<init>`), they only differ in signature and thus our technique may become too imprecise, lacking this core piece of distinguishing information. To address this issue, we employ a heuristic: we only accept calls to Java constructors from native code, if instances of the constructed type are used. We assume that  $type$  is used if an instance method is already reachable or if an instance field is accessed (Figure 10). This heuristic is successful in practice, since native code constructs objects that will be passed to the Java program to be actually used there. The result of this filtering step, which runs in mutually recursive fashion with the rest of the analysis, is predicate HighlyProbableNativeCodeTargetMethod(method, function, file).

**4.3.3 Marking Native Allocations.** Native allocations are easy to spot: creating an object in native code needs the native code to call a Java constructor method, so we can record such allocations in predicate NativeAllocation(constructor, function, file, type), where a constructor method is called from a function in a library file, to construct some type (Figure 11). As a detail of the Java implementation, we also model the reachability of a method in the library class `java.lang.ref.FinalizerReference`.

```
// A filtered version of the cross-product.
.decl HighlyProbableNativeCodeTargetMethod(method:Method, function:symbol, file:symbol)
// Accept calls to non-constructors.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
    PossibleNativeCodeTargetMethod(method, function, file),
    !ClassConstructor(method, _).

// Accept calls to constructors if instance methods are reachable.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
    PossibleNativeCodeTargetMethod(method, function, file),
    ClassConstructor(method, type),
    Method_DeclaringType(instanceMethod, type),
    Reachable(instanceMethod),
    !Method_Modifier("final", instanceMethod).

// Accept calls to constructors if instance fields are used.
HighlyProbableNativeCodeTargetMethod(method, function, file) :-
    PossibleNativeCodeTargetMethod(method, function, file),
    ClassConstructor(method, type),
    Field_DeclaringType(field, type),
    ( StoreHeapInstanceField(field, _, _, _, _, _)
    ; LoadHeapInstanceField(_, _, field, _, _, _)
    ), !Field_Modifier("final", field).
```

**Figure 10: Filtering the cross-product for constructors.**

```
// Native allocations detected by the native scanner.
.decl NativeAllocation(allocator:Method, function:symbol, file:symbol, type:ReferenceType)

NativeAllocation(allocator, function, file, type) :-
    HighlyProbableNativeCodeTargetMethod(allocator, function, file),
    ClassConstructor(allocator, type).

// Native allocations trigger finalization code in the Android runtime. See:
// https://android.googlesource.com/platform/libcore/+/master/luni/src/main/java/java/lang/ref/FinalizerReference.java
ReachableMethodFromNativeCode(finalizerAdd) :-
    NativeAllocation(_, _, _, _),
    finalizerAdd = "<java.lang.ref.FinalizerReference: void add(java.lang.Object)>",
    isMethod(finalizerAdd).
```

**Figure 11: Native allocations.**

```
static const char *className = "jackpal/androidterm/compat/FileCompat$Api80rEarlier";
static JNINativeMethod method_table[] = {
    { "testExecute", "(Ljava/lang/String;)Z", (void *) testExecute },
};

int init_FileCompat(JNIEnv *env) {
    if (!registerNativeMethods(env, className, method_table,
        sizeof(method_table) / sizeof(method_table[0]))) {
        return JNI_FALSE;
    }

    return JNI_TRUE;
}
```

**Figure 12: Example use of JNI function RegisterNatives().**

#### 4.4 Step EDGE

Knowing the native function that calls a Java method  $m$  is useful but we would like to also know if this function implements some Java method  $m'$  marked with the native keyword. If that is the case, then we should inform the pure-Java call graph and build

an edge from  $m'$  to  $m$ . Step EDGE takes care of this computation by observing how JNI links native code functions to Java native method entry points.

There are essentially two approaches to this linking: (a) the automatic one, where the default JNI behavior automatically connects

native and Java code, and (b) the programmable one, where the JNI code itself configures these entry points.

**4.4.1 Automatic Linking.** The automatic linking is easy to handle (and model as logic): the JNI specification follows a default naming convention that links a Java method with this declaration:

```
package x.y;
class C {
    native meth(Object obj);
}
```

with function `Java_x_y_C_meth`, possibly with a signature suffix in the case of method overloading.

**4.4.2 Configurable Linking.** The JNI specification offers a method `RegisterNatives()` that allows code to programmatically link native methods against Java “native” method entry points [29, Section 8.3]. This functionality can be useful for overriding the default JNI naming convention for native functions as shown in Figure 12 (code taken from the Android-Terminal-Emulator Android application<sup>5</sup>). Another use of JNI programmable linking is to allow for relinking different native methods to the same Java method.

We observe that some JNI uses of `RegisterNatives()` depend on strings and structures defined outside functions, which go into a different section in the binary code (and gathered in step FILT).

On Android, applications may have to resort to a custom linker (“crazy linker”<sup>6</sup>) to allow for flexibility in finding native code and portability across different Android versions. In particular, the default Web browser on Android (Chrome) and its assorted system framework (WebKit)<sup>7</sup> depend on such configurable linking [19].

`RegisterNatives()` works by accepting triplets (name, signature, function), where name and signature describe the Java method and function is a pointer to the native function. Our technique can recover the data structures containing these triplets, for native code that is amenable to function boundary analysis and implicit-jump analysis.

## 5 DISCUSSION

This section discusses pragmatics concerning our technique’s application: we show how to integrate with optimization tools used in Android development (Section 5.1), how the approach works in context-sensitive settings (Section 5.2), and the preconditions as well as variations of its applicability (Section 5.3).

### 5.1 Example Client Analysis: Code Optimization

An application of our technique (which we are actively exploring in a tool for wide use) is in the context of optimization of Android applications (“apps”). Android apps often use special tools (the two most common being ProGuard [23] and R8 [20]) to optimize the code that will be shipped to the “app store” and will then be downloaded by users. Two core tasks carried out by these tools are:

<sup>5</sup><https://github.com/jackpal/Android-Terminal-Emulator>

<sup>6</sup>[https://chromium.googlesource.com/android\\_tools/+/53862978424412e190e9bc40c7637a71fdd7d298/ndk/sources/android/crazy\\_linker/README.TXT](https://chromium.googlesource.com/android_tools/+/53862978424412e190e9bc40c7637a71fdd7d298/ndk/sources/android/crazy_linker/README.TXT)

<sup>7</sup><https://developer.chrome.com/multidevice/webview/overview>

- (1) *Code shrinking*: Android app binaries are big by design, since they have to include any library they may use, if it is not part of the standard platform. Since smaller app sizes correlate with higher install conversion rates [41], this optimization is crucial.
- (2) *Code obfuscation*: regardless of application size, Android developers often want to discourage reverse engineering of their code and thus resort to code obfuscation, even for small programs. As Wermke *et al.* found, obfuscation is performed in roughly 25% of apps in general and in 50% of the most popular apps [45]).

Both of the above program transformations cannot happen automatically for the case of native code calling Java code, since a Java-only analysis cannot find methods called from native libraries. Such missed methods will be marked as dead code and eliminated by the code shrinking phase or will be missed as entry points and renamed by the obfuscating phase; the outcome in both cases is run-time crashes due to missing methods.

Since these optimizations cannot happen automatically, both ProGuard and R8 use a configuration language that allows the developer to manually mark code that should not be eliminated or obfuscated. Maintaining these manual scripts is often a difficult and delicate process [45].

Our technique helps by automatically generating such configuration rules to be fed to the optimization tool, without the developer having to reason about the native code bundled in the application. We have implemented such a client analysis in Doop:<sup>8</sup> this is a reachability analysis that receives the configuration rules of an Android app and merges their effects with the results computed by the technique presented in this paper.

### 5.2 Context Sensitivity

Our approach fixes unsoundness in reachability metrics by finding entry points from native code. So far, these native-to-Java calls are assumed to happen in a context-insensitive way: the native code calls Java code using a hard-coded (constant) context.

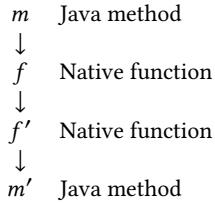
Using our technique in a context-sensitive analysis requires that an existing context be propagated, which assumes the presence of call-graph information for the native code. Assuming that the native code is not hostile or stripped, Radare2 can compute this call graph and we can use the EDGE step (Section 4.4) to connect invocation contexts to native code, via this call graph, to the callback invocations to Java code. Thus, context sensitivity is fully supported for native code that is amenable to call-graph analysis.

In the case of native code that is not amenable to the above analysis, we can still compute hard-coded contexts for callbacks, without breaking the context-sensitive analysis of the rest of the code. We believe that having hard-coded contexts for *difficult programs* under context sensitivity is a reasonable best-effort approach.

### 5.3 Applicability

The approach described identifies entry points, where native code can invoke Java code. Importantly, the approach does not aim to recover a full native call-graph: calling patterns inside native code are not tracked. Assume the following call chain:

<sup>8</sup>Option --keep-spec in Doop 4.20.46.



The call from  $m$  to  $f$  is a JNI call to native code (uncovered by the EDGE step), while the call from  $f'$  to  $m'$  is a call-back (uncovered by the LOC step). For the EDGE step to find that  $m$  eventually calls  $m'$ , it must be known that  $f$  calls  $f'$ . Our approach does not aim to recover such call-graph edges. However, such call-graph information can be extracted from Radare2, as long as native code is not hostile (i.e., obfuscated) or stripped. It is important to note that ignoring native call-graph edges does not affect the completeness of our technique: the Java entry point will be identified regardless.

Our approach does not depend on Android, Linux, or ELF, but works for every platform where an appropriate disassembler exists, allowing automatic discovery of referenced constant strings.

Our technique can be generalized to catch other string-based JNI functionality such as field reads or writes, to fix unsoundness in points-to analyses. Our approach can also be generalized to other programming languages that declare a foreign function interface (FFI) to native code that uses string constants for interoperability with the high-level language.

Finally, note that we do not attempt to detect the names of the enclosing classes of the callback methods. In our manual inspection, we see that method names and signatures are mainly constants in the native code. The classes or interfaces containing these methods are what is sometimes a parameter or a dynamic value (e.g., read from serialization data) and thus we choose to not rely on native code strings for type information.

*Restrictions.* Handling the general case of configurable linking (i.e., calls to `RegisterNatives()`) may fail, especially for stripped native code, where function pointers can go through relocations [27, Chapter 8].

Additionally, we do not handle JNI code that calls methods using dynamically-generated strings or interoperability with Java reflective method values (such as `FromReflectedMethod` [31, Chapter 4]).

*Handling Uncooperative Native Code.* We recap by accumulating, for reference purposes, all parts of our technique that may be affected by native code that is stripped, hostile, obfuscated, or otherwise not suitable for string x-refs analysis or call-graph analysis:

- configurable linking (Section 4.4.2)
- localization of call-back invocations (Section 4.3)
- context sensitivity (Section 5.2)

In handling these elements, our technique may lose precision but will not miss a call-back target that it would otherwise find. Notably, the string x-refs analysis (which attempts to compute which native functions use which strings) has a conservative fallback. If a string is found to not be referenced by any function, we give it a default wildcard function, so that it will still be considered, albeit imprecisely.

## 6 EVALUATION

We evaluate our analysis on an independently-selected set of large Java programs, both desktop and Android. The programs include the subset of the XCorpus suite that contains native code (Section 6.1) and large Android applications (Section 6.2).

We integrated our analysis in Doop [5], version 4.20.46.<sup>9</sup> All experiments are run on a 64-bit machine with an Intel Xeon CPU E5-2667 v2 3.30GHz with 256 GB of RAM. We use the Soufflé compiler (v.1.5.1), which compiles Datalog specifications into binaries via C++ and run the resulting binaries in parallel mode using four jobs. Doop uses the Java 8 platform as implemented in Oracle JDK v1.8.0\_121. All metrics are for Doop’s default context-insensitive analysis.

In our experiments, we measure reachable methods from native code (that would be missed by a naive analysis) in the application, i.e., calls from native libraries bundled with the program, and not native code in the JDK or Android platform. The platform libraries certainly also contain native code, but since they are the same for all applications their behavior can be modeled explicitly. For instance, Doop already contains explicit models for a variety of JDK native methods, for the Android app lifecycle, as well as for frameworks involving native code, such as reflection [36] and `invokedynamic` [16].

### 6.1 XCorpus

XCorpus is a suite of executable Java programs containing features that are difficult to analyze by current tools [11]. We focus on the four benchmarks that the “feature analysis” tool of XCorpus reports as having native code [11, Table 3]: *aspectj-1.6.9*, *log4j-1.2.16*, *lucene-4.3.0*, and *tomcat-7.0.2*. We manually inspected the sources of each benchmark.<sup>10</sup>

- *aspectj* uses native code for filesystem functionality on Windows (example code: Figure 13).
- *log4j* contains native code that does not call back to Java code.
- *lucene*: on POSIX-style systems, native code constructs and returns a file descriptor object (example code: Figure 14).
- *tomcat* uses a native library for performance [15] (example code: Figure 15).

We found that our technique captures all call-back targets from the native code. Figure 16 shows the results of analyzing these four XCorpus benchmarks. For every program, we calculate the total number of application methods (*App methods*) and the increase in their reachability by our technique (*+App-reachable*). This increase is due to our scanning introducing a number of candidate Java methods as call-backs (*+Entry points*), which are added to the rest of the analysis as additional entry points for further analysis. We also measure the impact of our technique on analysis time (*+Analysis time*) and on fact generation time (*+Factgen time*). That last metric, fact generation time, includes the initial processing of the native library code (steps PRE, FILT, and LOC, described in Section 4).

<sup>9</sup>Our technique is freely available as (a) a front end (<https://github.com/plast-lab/native-scanner>) that extracts appropriate information from Java programs and (b) Datalog rules in the Doop framework (<https://bitbucket.org/yanniss/doop>).

<sup>10</sup>Available in <https://bitbucket.org/gfour/xcorpus-native-extension>.

```

/*
 * Converts a WIN32_FIND_DATA to IFileInfo
 */
jboolean convertFindDataToFileInfo(JNIEnv *env, WIN32_FIND_DATA info, jobject fileInfo) {
    ...
    // select interesting information
    //exists
    mid = (*env)->GetMethodID(env, cls, "setExists", "(Z)V");
    if (mid == 0) return JNI_FALSE;
    (*env)->CallVoidMethod(env, fileInfo, mid, JNI_TRUE);

    // file name
    mid = (*env)->GetMethodID(env, cls, "setName", "(Ljava/lang/String;)V");
    if (mid == 0) return JNI_FALSE;
    (*env)->CallVoidMethod(env, fileInfo, mid, windowsTojstring(env, info.cFileName));

    // last modified
    mid = (*env)->GetMethodID(env, cls, "setLastModified", "(J)V");
    if (mid == 0) return JNI_FALSE;
    (*env)->CallVoidMethod(env, fileInfo, mid, fileTimeToMillis(info.ftLastWriteTime));
}

```

**Figure 13: Native code in AspectJ (from library *eclipse.platform.resources*).**

```

/*
 * Class:      org_apache_lucene_store_NativePosixUtil
 * Method:     open_direct
 * Signature:  (Ljava/lang/String;Z)Ljava/io/FileDescriptor;
 */
extern "C"
JNIEXPORT jobject JNICALL Java_org_apache_lucene_store_NativePosixUtil_open_1direct(JNIEnv *env, jclass _ignore, jstring filename, jboolean readOnly)
{ ...
    class_fdesc = env->FindClass("java/io/FileDescriptor"); ...
    // construct a new FileDescriptor
    const_fdesc = env->GetMethodID(class_fdesc, "<init>", "()V"); ...
    ret = env->NewObject(class_fdesc, const_fdesc);

    // poke the "fd" field with the file descriptor
    field_fd = env->GetFieldID(class_fdesc, "fd", "I"); ...
    env->SetIntField(ret, field_fd, fd);

    // and return it
    return ret;
}

```

**Figure 14: Native allocation in Lucene.**

Running times are for a single run, hence noisy. (A 5% variation between runs is common, in our experience.) Establishing statistically significant precision in running times is, however, far from the point: we intend to observe potential order-of-magnitude differences, not small variations. As seen, the extra processing is usually cheap, unless the program has a lot of binary code; in that case, it is close to the initial processing step for Java bytecode (and these two steps can happen in parallel, as they reason about different parts of the application).

We also see that our technique does not introduce noise in the benchmark where no native call-backs are to be found (*log4j*).

## 6.2 Android Applications

We evaluate (Figure 17) our technique on the Android apps from the benchmark suite of the HeapDL tool [21], which produces snapshots of the dynamic activity of Java applications. HeapDL is integrated with Doop and its published benchmarks<sup>11</sup> contain popular and complex Android apps (Chrome, Instagram, Google Translate, pinterest, S PhotoEditor, androidterm). We analyzed the dynamic activity logs of these apps and two of them (chrome and Instagram) exhibit call backs from native code to Java code. (It is likely that the rest of the apps also perform such call backs, but they are not exercised in the HeapDL dynamic executions.)

<sup>11</sup><https://bitbucket.org/yanniss/doop-benchmarks>

```
#define TCN_IMPLEMENT_CALL(RT, CL, FN)    JNIEXPORT RT JNICALL Java_org_apache_tomcat_jni_##CL##_##FN

TCN_IMPLEMENT_CALL(jlong, Pool, cleanupRegister)(TCN_STDARGS, jlong pool, jobject obj)
{ ...
    cls = (*e)->GetObjectClass(e, obj);
    cb->obj      = (*e)->NewGlobalRef(e, obj);
    cb->mid[0] = (*e)->GetMethodID(e, cls, "callback", "()I");

    apr_pool_cleanup_register(p, (const void *)cb, generic_pool_cleanup, apr_pool_cleanup_null); ...
}

TCN_IMPLEMENT_CALL(jlong, SSL, newBIO)(TCN_STDARGS, jlong pool, jobject callback)
{ ...
    j = (BIO_JAVA *)BIO_get_data(bio); ...
    cls = (*e)->GetObjectClass(e, callback);
    j->cb.mid[0] = (*e)->GetMethodID(e, cls, "write", "([B)I");
    j->cb.mid[1] = (*e)->GetMethodID(e, cls, "read",   "([B)I");
    j->cb.mid[2] = (*e)->GetMethodID(e, cls, "puts",   "(Ljava/lang/String;)I");
    j->cb.mid[3] = (*e)->GetMethodID(e, cls, "gets",   "(I)Ljava/lang/String;");
    j->cb.obj     = (*e)->NewGlobalRef(e, callback); ...
}
```

Figure 15: Native code in Tomcat.

Benchmark	App methods (native)	+App-reachable	+Analysis time	+Factgen time	+Entry points
aspectj-1.6.9	41749 (8)	13034→13454: 3.22%	229→249: 8.7%	74→78: 5.4%	47
log4j-1.2.16	3423 (3)	961→961: 0.00%	60→58: -3.3%	47→49: 4.3%	0
lucene-4.3.0	33393 (9)	12414→12729: 2.54%	108→291: 169.4%	55→56: 1.8%	295
tomcat-7.0.2	19661 (273)	1203→2088: 73.57%	59→218: 269.4%	61→95: 55.7%	308

Figure 16: XCorpus benchmarks.

Benchmark	App meths (native)	Base recall	Recall	+App-reachable	+Analysis time	+Factgen time	+Entry points
Chrome	37898 (1531)	7/83 = 8.43%	83/83 = 100.00%	17003→24060: 41.50%	469→505: 7.7%	46→255: 454.4%	4484
Instagram	43420 (348)	1/7 = 14.29%	7/7 = 100.00%	23921→32425: 35.55%	473→625: 32.1%	51→63: 23.5%	4669

Figure 17: Android apps.

Our technique exhibits a 100% recall rate over the observed dynamic call-backs from native to Java code. This includes covering the full set of 83 dynamically-observed entry points in Chrome. As a result of adding these entry points, a much larger part of the application code becomes analysis-reachable. Accordingly, we also see a big increase in the fact generation time for Chrome: this is due to our Radare2 back-end requiring significant time to parse and analyze the bundled native library.

## 7 RELATED WORK

Our work connects declarative static analysis with information coming from a reverse engineering front end (Radare2), running on native code. Another declarative analysis tool working on native code is *ddisasm*, developed by GrammaTech, Inc. [13]. This framework also uses Datalog and achieves a high-level of accuracy, being capable to correctly reassemble its disassembled output. Our

choice of Radare2 over ddisasm as a front end was pragmatic: ddisasm only supports the x86-64 platform,<sup>12</sup> while we require mature analysis on x86 32-bit code and the Android ARM targets (arm64 and armeabi [18]).

Redex is an open source Android bytecode optimizer developed by Facebook [25]. To improve performance and efficiency of Android apps, Redex applies optimizations such as dead code elimination, inlining, and minification, and removing unnecessary metadata. Redex also scans native libraries to find class names,<sup>13</sup> although it does not go as far as our technique to discover method call-backs and native allocations.

Although well-designed to balance security checks while allowing for low-level performance, the JNI is still a source of vulnerabilities [22, 28, 37, 40] due to native code being opaque and thus less amenable to static analysis. This has led in practice to a multitude of JNI sandboxes (some even in hardware) to contain the effects of native code running alongside Java code [6, 39].

<sup>12</sup><https://github.com/GrammaTech/ddisasm/issues/2>

<sup>13</sup><https://github.com/facebook/redex/blob/master/libredex/RedexResources.cpp>

Recently, Wei *et al.* showed how to do cross-language dataflow-based security analysis on Android applications that contain native code [43]. Similar to our approach, they model parts of the JNI API and use a reverse engineering front end (*angr* [35]). Their native code modeling is based on symbolic execution, as opposed to static recognition of JNI strings. This is a good fit for the intended goal of the approach: getting a precise model of native behavior, for security analysis purposes [44]. In contrast, our technique emphasizes more complete analysis, integrating generally with all existing Doop analyses, yet without a precise model of information flow through native code.

Furr and Foster also analyzed JNI strings in binaries for a different reason: type inference over the JNI boundary [17]. They also worked on the level of C sources, not compiled binaries. However, they “found that simple tracking of strings is sufficient”, which is also a core idea of our approach.

Our analysis can be combined with other JNI-based analyses such as the security analysis of Li and Tan [28].

## 8 CONCLUSION

We showed that simple declarative analysis logic can be coupled with filtering logic to work on fixing false negatives in static reachability analysis. Our technique also uses an existing reverse engineering front end that does string x-refs analysis for added precision and reconstruction of programmable linking configurations.

Our technique is lightweight but can be a starting point for the incorporation of heavier static analyses on native code [3, 4, 34, 35]. Such tools (plus additional string analysis [7]) can offer more insight into specific uses of the JNI specification, such as handling JNI uses with on-the-fly construction of strings.

## ACKNOWLEDGMENTS

We gratefully acknowledge support by the Hellenic Foundation for Research and Innovation, grant DEAN-BLOCK.

## REFERENCES

- [1] Sumaya Almanee, Mathias Payer, and Joshua Garcia. 2019. Too Quiet in the Library: A Study of Native Third-Party Libraries in Android. [arXiv:cs.CR/1911.09716](https://arxiv.org/abs/cs.CR/1911.09716)
- [2] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 583–600.
- [3] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Compiler Construction*, Rastislav Bodík (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–254.
- [4] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–104.
- [5] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. <https://doi.org/10.1145/1640089.1640108>
- [6] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, and et al. 2017. CHERI JNI: Sinking the Java Security Model into the C. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 569–583. <https://doi.org/10.1145/3037697.3037725>
- [7] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. 2005. String Analysis for x86 Binaries. (2005), 88–95. <https://doi.org/10.1145/1108792.1108814>
- [8] DWARF Standards Committee. [n. d.]. The DWARF Debugging Standard. <http://dwarfstd.org/>.
- [9] Radare2 Contributors. 2020. ESIL - Radare2 Book. <https://radare.gitbooks.io/radare2book/disassembling/esil.html>.
- [10] WALA Developers. 2019. TJ. Watson Libraries for Analysis (WALA). <http://wala.sourceforge.net>.
- [11] Jens Dietrich, Henrik Schole, Li Sui, and Ewan D. Tempero. 2017. XCorpus - An executable Corpus of Java Programs. *Journal of Object Technology* 16, 4 (2017), 11–24. <https://doi.org/10.5381/jot.2017.16.4.a1>
- [12] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. 2017. On the Construction of Soundness Oracles. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 37–42. <https://doi.org/10.1145/3088515.3088520>
- [13] Antonio Flores-Montoya and Eric M. Schulte. 2019. Datalog Disassembly. *CoRR* abs/1906.03969 (2019). arXiv:1906.03969 <http://arxiv.org/abs/1906.03969>
- [14] Free Software Foundation. 2017. GNU Binutils. <https://www.gnu.org/software/binutils/>.
- [15] The Apache Software Foundation. 2019. Apache Tomcat Native Library - Documentation Index. <http://tomcat.apache.org/native-doc>.
- [16] George Fourtounis and Yannis Smaragdakis. 2019. Deep Static Modeling of invoke-dynamic. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15–19, 2019, London, United Kingdom (LIPIcs)*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 15:1–15:28. <https://doi.org/10.4230/LIPIcs.ECOOP.2019.15>
- [17] Michael Furr and Jeffrey S. Foster. 2006. Polymorphic Type Inference for the JNI. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–324.
- [18] Google. 2020. Android ABIs - Android NDK - Android Developers. <https://developer.android.com/ndk/guides/abis>.
- [19] Google. 2020. Shared Libraries on Android. [https://chromium.googlesource.com/chromium/src/+/master/docs/android\\_native\\_libraries.md](https://chromium.googlesource.com/chromium/src/+/master/docs/android_native_libraries.md).
- [20] Google. 2020. Shrink, obfuscate, and optimize your app | Android Developers. <https://developer.android.com/studio/build/shrink-code>.
- [21] Neville Grech, George Fourtounis, Adrián Francalanza, and Yannis Smaragdakis. 2017. Heaps Don’t Lie: Countering Unsoundness with Heap Snapshots. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 68 (Oct. 2017), 27 pages. <https://doi.org/10.1145/3133892>
- [22] Y. Gu, K. Sun, P. Su, Q. Li, Y. Lu, L. Ying, and D. Feng. 2017. JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 427–438. <https://doi.org/10.1109/DSN.2017.40>
- [23] Guardsquare. 2020. ProGuard - Official website - Java and Android Apps optimizer. <https://www.guardsquare.com/en/products/proguard>.
- [24] Chua Hock-Chuan. 2018. Java Native Interface Tutorial. <https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html>.
- [25] Facebook Inc. 2019. Redex - An Android Bytecode Optimizer. <https://fbredex.com/>.
- [26] Sungho Lee. 2019. JNI Program Analysis with Automatically Extracted C Semantic Summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 448–451. <https://doi.org/10.1145/329382.3338990>
- [27] John R. Levine. 1999. *Linkers and Loaders* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [28] Siliang Li and Gang Tan. 2009. Finding Bugs in Exceptional Situations of JNI Programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. Association for Computing Machinery, New York, NY, USA, 442–452. <https://doi.org/10.1145/1653662.1653716>
- [29] Sheng Liang. 1999. *Java Native Interface: Programmer’s Guide and Specification* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.
- [30] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. <https://doi.org/10.1145/2644805>
- [31] Oracle. 2020. Java Native Interface Specification Contents. <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jnTOC.html>.
- [32] OSDev.org. 2019. ELF (Executable and Linkable Format). <https://wiki.osdev.org/ELF>.
- [33] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. <https://doi.org/10.1145/329382.3330555>
- [34] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

- [35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
- [36] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 485–503. [https://doi.org/10.1007/978-3-319-26529-2\\_26](https://doi.org/10.1007/978-3-319-26529-2_26)
- [37] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjad Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.
- [38] Li Sui, Jens Dietrich, Amjad Tahir, and George Fournousis. 2020. On the Recall of Static Call Graph Construction in Practice. To appear in ICSE 2020.
- [39] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting Android Applications from Third-Party Native Libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14)*. Association for Computing Machinery, New York, NY, USA, 165–176. <https://doi.org/10.1145/2627393.2627396>
- [40] Gang Tan, Srimat Chakradhar, Raghunathan Srivaths, and Ravi Daniel Wang. 2006. Safe Java native interface. In *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*. 97–106.
- [41] Sam Tolomei. 2017. Shrinking APKs, growing installs – How your app’s APK size impacts install conversion rates. <https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2>. (Nov. 2017).
- [42] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13.
- [43] Fengguo Wei, Xingwei Lin, Ximeng Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-Aware Inter-Language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1137–1150. <https://doi.org/10.1145/3243734.3243835>
- [44] Fengguo Wei, Sankardas Roy, Ximeng Ou, and Robby. 2018. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security* 21, 3, Article 14 (April 2018), 32 pages. <https://doi.org/10.1145/3183575>
- [45] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 222–235. <https://doi.org/10.1145/3274694.3274726>

# An Empirical Study on ARM Disassembly Tools

Muhui Jiang

csmjiang@comp.polyu.edu.hk  
The Hong Kong Polytechnic  
University  
China

Ruoyu Wang

fishw@asu.edu  
Arizona State University  
USA

Yajin Zhou\*

yajin\_zhou@zju.edu.cn  
Zhejiang University  
China

Xiapu Luo

csxluo@comp.polyu.edu.hk  
The Hong Kong Polytechnic  
University  
China

Yang Liu

yangliu@ntu.edu.sg  
Nanyang Technological University  
Singapore  
Institute of Computing Innovation,  
Zhejiang University  
China

Kui Ren

kuiрен@zju.edu.cn  
Zhejiang University  
China

## ABSTRACT

With the increasing popularity of embedded devices, ARM is becoming the dominant architecture for them. In the meanwhile, there is a pressing need to perform security assessments for these devices. Due to different types of peripherals, it is challenging to dynamically run the firmware of these devices in an emulated environment. Therefore, the static analysis is still commonly used. Existing work usually leverages off-the-shelf tools to disassemble stripped ARM binaries and (implicitly) assume that reliable disassembling binaries and function recognition are solved problems. However, whether this assumption really holds is unknown.

In this paper, we conduct the first comprehensive study on ARM disassembly tools. Specifically, we build 1,896 ARM binaries (including 248 obfuscated ones) with different compilers, compiling options, and obfuscation methods. We then evaluate them using eight state-of-the-art ARM disassembly tools (including both commercial and noncommercial ones) on their capabilities to locate instructions and function boundaries. These two are fundamental ones, which are leveraged to build other primitives. Our work reveals some observations that have not been systematically summarized and/or confirmed. For instance, we find that the existence of both ARM and Thumb instruction sets, and the reuse of the *BL* instruction for both function calls and branches bring serious challenges to disassembly tools. Our evaluation sheds light on the limitations of state-of-the-art disassembly tools and points out potential directions for improvement. To engage the community, we release the data set, and the related scripts at [https://github.com/valour01/arm\\_disassembler\\_study](https://github.com/valour01/arm_disassembler_study).

\*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397377>

## CCS CONCEPTS

• Software and its engineering → Assembly languages.

## KEYWORDS

Disassembly Tools, ARM Architecture, Empirical Study

### ACM Reference Format:

Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397377>

## 1 INTRODUCTION

ARM is becoming the dominant architecture for embedded and mobile devices. At the same time, the security of these devices has attracted raising attentions [26, 32, 34, 41, 57, 59], possibly due to serious consequences if they are compromised. For instance, a botnet with hijacked IoT devices could bring down popular websites [26] and disrupt power grids [59].

The software of these devices, i.e., the firmware, is usually closed-source. Researchers have to analyze the (stripped) firmware without access to the source code or debugging symbols. Due to different types of peripherals of these devices, it is an ongoing research problem to dynamically run firmware under an emulated environment at a large scale [33, 34, 39, 65]. Because of this (sad) fact, static analysis is still a dominating methodology used by the community [32, 35, 40, 57, 63]. For instance, it has been used to locate bugs [40], find authentication bypass vulnerabilities<sup>1</sup> [57] and build a general framework to rewrite ARM binaries<sup>1</sup> [60].

As shown in Table 1, previous systems usually leverage off-the-shelf disassembly tools to identify instructions and function boundaries. They assume that reliably disassembling stripped binaries is a solved problem. However, whether this assumption really holds is unknown. Andriesse et al. [25] performed an analysis of disassembly tools on *x86/x64 binaries* and presented their findings that “*some constructs, such as function boundaries, are much harder*

<sup>1</sup>In this paper, we use “ARM binaries” to refer to binary programs running on CPUs of the ARM architecture. An ARM binary can have both ARM and Thumb instruction sets (Section 2.1) [32].

**Table 1: A summary of representative research prototypes and their supported primitives and used disassembly tools.**

System	Instruction Boundary	Function Boundary	Control Flow Graph	Call Graph	Disassembly Tools
Firmalice[57]	✓	✓	✓	✓	angr
Firmup[35]	✓	✓	✓		IDA Pro
Genius[40]	✓	✓	✓		IDA Pro
Gemini[63]	✓	✓	✓		IDA Pro
RevARM[60]	✓	✓	✓		IDA Pro
Bug Search[51]	✓		✓		IDA Pro
discovRE[38]	✓	✓	✓	✓	IDA Pro
C-FLAT[24]	✓		✓		Capstone

to recover accurately than is reflected in the literature.” Although the paper provides insights of the mismatch between assumptions in papers and current capabilities of popular disassembly tools, it only focuses on x86/x64 binaries. Whether their findings could be applied to ARM binaries is unknown.

**Challenges of disassembling ARM binaries** ARM binaries have some unique properties, which bring challenges when disassembling them. First, inline data is common in ARM binaries while “constructs like inline data and overlapping code are very rare” in x86/x64 binaries [25]. Second, ARM provides two instruction sets: the ARM instruction set and the Thumb instruction set (which includes both 16-bit Thumb-1 and 32-bit Thumb-2 instructions). An ARM binary can have both ARM and Thumb instructions. Precisely identifying the correct instruction set is challenging. Third, there is no distinguished function call instruction in ARM binaries, unlike the `call` instruction on x86/x64. Compilers tend to reuse other instructions, such as the branch and link instruction (`BL`), for both function calls and direct branches in the Thumb instruction set. This makes identifying functions more challenging. Due to these unique properties, there is a need to perform an extensive and thorough evaluation of ARM disassembly tools.

**Our work** In this work, we perform an empirical study of ARM disassembly tools. In particular, we evaluate these tools’ capabilities of two primitives, the instruction boundary and the function boundary. These two are fundamental ones that other primitives (e.g., control flow graph and call graph) are built upon. To make the study comprehensive, it should meet the following requirements:

- (1) Our evaluation should use diverse programs, including both popular benchmarks and, more importantly, different types of real programs that are commonly used in the wild.
- (2) Our evaluation should cover different compilers with various compiling options, e.g., different instruction sets and optimization levels.
- (3) Our evaluation should consider options of tools, which may affect the result.
- (4) Our evaluation should include obfuscated binaries [67], since they do exist in the wild and could affect the results of disassembly tools.

To this end, we cross-compile 1,040 real-world programs and 19 benchmark programs. In total, we get 1,896 binaries, where 608 are from the SPEC CPU2006 with different compiling options. Among the remaining ones, 1,040 are Android daemons, libraries, and user-space binaries of embedded systems (e.g., OpenWRT [19]). We also build 248 obfuscated binaries using O-LLVM [43] with

multiple obfuscation methods. Then we obtain the ground truth with the help of debugging symbols and feed the stripped binaries (binaries without debugging symbols) to eight state-of-the-art ARM disassembly tools, including three commercial ones (i.e., IDA Pro [13], Hopper [12], and Binary Ninja [5]) and five noncommercial ones (i.e., Ghidra [11], arm-linux-gnueabi-objdump [18], angr [58], Radare2 [22], and BAP [30]). Finally, we measure the precision and recall by comparing the differences between the ground truth and the disassembling result.

Based on the result, we present some observations that were not systematically summarized and/or confirmed. First, the unique properties of ARM binaries do bring challenges to the disassembly tools, especially the two different instruction sets (i.e., the ARM instruction set and the Thumb instruction set) and the reuse of the `BL` label for both function call and branch. Second, disassembly tools do not have a good support to binaries in Thumb instruction set. The precision and recall for disassembling Thumb instructions are usually lower than that of ARM instructions (more than 90% in maximum). Third, the robustness and scalability of disassembly tools should be improved. We observed several exceptions, segment faults and timeout during the analysis. Fourth, other factors, including compilers, compiling options, target CPU architectures, could affect the result. However, the root cause is still due to the unique properties of ARM binaries.

We have reported our findings along with failed test cases to developers of the evaluated tools [14–17]. Developers of Binary Ninja, Hopper, and angr verified our findings and provided updates based on the failed cases. Radare2 assigned bug tag to them. Ghidra verified our findings and provided the potential solutions, while BAP declared that they would solve the problem in the future. To engage the community, we release the data set, and the related scripts at [https://github.com/valour01/arm\\_disassembler\\_study](https://github.com/valour01/arm_disassembler_study).

In summary, this work makes the following contributions.

- We summarize the unique properties of ARM binaries that bring challenges to the disassembling of ARM binaries.
- We perform the first comprehensive study of state-of-the-art disassembly tools on ARM binaries and report our findings, which show that, contrary to the previous assumption, reliably disassembling ARM binaries is not yet a solved problem.
- Our evaluation and further analysis of failed cases reveal their root causes and provide insights and future directions for improvements.

## 2 BACKGROUND

### 2.1 CPU Architectures and Instruction Sets

ARM has multiple CPU architectures, each with different instruction extensions and features. When building binaries, developers can specify the target CPU architecture, e.g., ARMv5 or ARMv7, through compiling options (`-march`). For instance, ARMv5 is the default CPU architecture of the GCC compiler.

Moreover, there are two instruction sets, i.e., the ARM instruction set and the Thumb instruction set. The former is 32-bit long, while the latter is 16-bit long and designed for size-sensitive applications, which is available for ARMv4T CPU architecture and later versions. Since ARMv6T2, Thumb-2 is introduced. It offers “best of both worlds” compromise between the ARM instruction set and the Thumb instruction set. It has access to both 16-bit and 32-bit

C	Thumb	ARM
<pre>uint8 foo(uint8 x, uint8 a,          uint16 b, uint16 c) {     if (a==2) x += (b &gt;&gt; 8);     else x += (c &gt;&gt; 8);     return x; }</pre>	<pre>0x00: 2002 CMP r1,#2 0x01: b1f4 ITTE 0x04: e000 BNE 0x06: e000 LSR r1,r2,#8 0x08: e000 LSR r1,r3,#8 0x0a: 1008 ADDS r0,r1,r0 0x0c: 0000 LSLS r0,r0,#24 0x0e: 0000 LSR r0,r0,#24 0x10: 4770 BX lr</pre>	<pre>0x00: e3510002 CMP r1,#2 0x01: 10000000 ADDNE r0,r0,r3,LSR #8 0x04: e0000002 ADDEQ r0,r0,r2,LSR #8 0x06: b0000000 BX r0,r0 0x08: 00000002 ADDEQ r0,r0,r2,LSR #8 0x0c: e00000ff AND BX r0,r0 0x10: e12ffffe BX lr</pre>
Thumb-2		

Figure 1: The source code and its corresponding ARM, Thumb and Thumb-2 instructions.

instructions. In this paper, we use the Thumb instruction set to denote both Thumb and Thumb-2 instruction encoding.

A single binary can contain multiple instruction sets and switch between them, e.g., switching between ARM instructions and Thumb (Thumb-2) instructions. The switching can occur explicitly by executing branch instructions or implicitly specified by branch targets. For instance, the `BLX label` instruction always changes the instruction set from ARM to Thumb or vice versa. However, the `BX Rm` derives the target instruction set from `bit[0]` of the register Rm. If it is 0, then the target instruction set is ARM. Otherwise, it is Thumb. The target instruction set of other branch instructions, e.g., `POP {PC, Rm ...}`, also depends on the last bit of the target address. This brings serious challenges for disassembly tools to *statically* determine the target instruction set, especially for the ones that leverage linear sweep strategy (Section 2.2).

Figure 1 illustrates the source code of a function and the binary compiled using ARM, Thumb and Thumb-2 instructions. Note that, for the two popular compilers (e.g., GCC and Clang), the default instruction set is ARM, and the option `-mthumb` is used to change it to Thumb. The instruction set greatly affects the accuracy of disassembly tool, which we will discuss in Section 4.

## 2.2 Disassembly Strategies

To understand the capability of disassembly tools, we use two primitives, i.e., the instruction boundary and the function boundary in our study. To precisely detect the instruction boundary, a disassembly tool should be able to locate the inline data inside the binary and the correct instruction set (ARM vs Thumb).

There are two different disassembly strategies [52, 55]. One is linear sweep, which linearly decodes the code sections. It is used by disassemblers such as the GNU utility `objdump`. However, the inline data (data inside the code section), and instruction set switching cannot be detected by this strategy since it does not consider the control flow transfers. Figure 2 shows a function with three basic blocks and inline data between the basic block 2 and the basic block 3. The `objdump` tool fails to determine the boundary between code and data, and disassembles the inline data as code.

Another strategy is recursive traversal. Its basic idea is disassembling code from the entry point of a binary, and then recording the branch targets as new entry points (usually appends these branch targets into a list). It repeats this process until no new targets could be found, and all the targets in the list have been traversed. The advantage of this strategy is that it is unlikely to disassemble inline

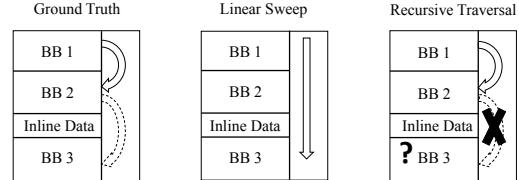


Figure 2: Two disassembly strategies. The function has three basic blocks (BBs), and inline data between BB2 and BB3. There is a direct jump from BB1 to BB2 and an indirect jump from BB2 to BB3.

data as code, since there should be a control flow transfer instruction before the inline data (otherwise, the data will be executed as code at runtime). Moreover, it can handle instruction set switch if the branch target can be determined statically (direct branches). However, the disadvantage is that some code regions may be missed, if they cannot be reached through direct branches. As in Figure 2, the code in the basic block 3 may be missed since this block can only be reached through an indirect branch, whose target is determined at runtime. Note that, even though methods [31] have been proposed to detect the targets of a jump table (one type of indirect branches), how to reliably detect other types of indirect branches (e.g., function pointers) is still an open research question. We do find that resolving indirect jump targets can improve the result of disassembly tools (Section 4.4.2).

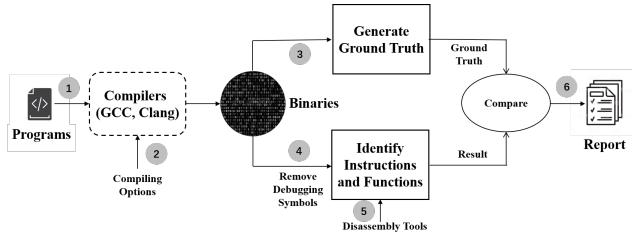
## 2.3 Function Identification

Function identification is an important primitive, which can be used to construct other primitives, e.g., function call graph. Previous work usually leverages function signatures to detect functions. The method proposed in [27, 30] scans a binary for known function prologues and epilogues. However, this method is limited by the fact that function signatures (e.g., prologues and epilogues) could be missing. Moreover, it's a tedious task to maintain a up-to-date signature database.

Due to these limitations, a compiler-agnostic function identification method was proposed in Nucleus [36]. The basic idea is to analyze the inter-procedural control flow graph (ICFG). To use Nucleus, the binary should have distinguished function call instructions, e.g., the `call` instruction for x86. Unfortunately, this assumption does not hold for the binary compiled using the Thumb instruction set. Specifically, the `BL label` instruction is intended for a direct function call. However, Thumb binaries reuse it for a direct branch (besides the function call) since the range of the branch target is larger than the `B label` instruction. As a result, it causes many false positives to the function detection. We observed a significant decrease of the precision value for the function boundary of binaries with the Thumb instruction set (Figure 8(b)).

## 2.4 Code Obfuscation

Some binaries are obfuscated. For instance, O-LLVM [43] is usually used to obfuscate programs. It supports the following obfuscation methods. Specifically, instructions substitution (sub) is used to replace standard operators with more complicated instruction sequences. The bogus control flow (bcf) changes a function's control



**Figure 3: Overview of our study**

flow graph by adding basic blocks. The control flow flattening (*f1a*) uses the control flow flatten algorithm [47] to create a large number of fake control flows. In this study, we use O-LLVM to generate the obfuscated binaries and feed them to disassembly tools to evaluate the impact of code obfuscation.

### 3 APPROACH

As shown in Figure 3, we first compile various types of programs, including popular benchmarks and real-world applications (①), using different compilers (GCC and Clang) with diverse compiling options (②). This aims to cover popular compilers and different scenarios that binaries are built with different options. After that, we first generate the ground truth by leveraging the debugging symbols (③), and then remove the symbols (④) and feed the stripped binaries to disassembly tools (⑤). We retrieve the result of identified instructions and functions for each tool, and compare the result with the ground truth (⑥) to generate the final report, which contains the recall and precision value. We present the main steps of our study in the following sections.

#### 3.1 Build Programs

One may think it is straightforward to compile binaries for evaluation. However, to make our study representative, we need to consider the types of programs and the diversity of compilers, compiling options, and obfuscation methods, which will affect the result.

**Different Types of Programs** We use three types of programs in our study. They include the widely used SPEC CPU2006, binaries in AOSP and OpenWRT. The latter two represent the binaries for mobile systems and IoT devices. Specifically, we compile the SPEC CPU2006 using both Clang and GCC compilers with two optimization levels (0s and 02), two instruction sets (ARM and Thumb) and two CPU target architectures (-march with ARMv5 and ARMv7). In total, we get 608 binaries, i.e., 19 benchmark programs  $\times$  2 instruction set  $\times$  2 optimization levels  $\times$  2 compilers  $\times$  (3 compiler versions + 1 specific CPU architecture with latest version of compilers) = 608 binaries.

Considering the popularity of IoT and mobile systems, we build the latest Android Open Source Project (AOSP version 9) and extract daemon binaries (127 in total) and libraries (667 in total). Also, we build the latest stable version of OpenWRT (version 18.06). There are 12 different target boards that support the ARM architecture. In total, we get 246 binaries.

**Compilers** We use two compilers, i.e., GCC and Clang, each with three different versions. Specifically, we use GCC versions 6.5, 7.5

and 8.3 and Clang versions 7.0, 8.0 and 9.0, which cover the major compilers used in the wild.

**Compiling Options** As mentioned in Section 2.1, different compiling options (e.g., with or without `-mthumb`) would result in completely different binaries. Considering the diversity of binaries, we aim to understand which compiling options are mostly used in the real world. Thus, we divide ARM binaries into three types, according to systems they are used.

- (1) **Type-I: Embedded OSes** They are used in resource-constrained ARM devices, mainly the ARM Cortex-M processor families with low computational power. We select FreeRTOS v10.1.1 [23], the most popular real time operating system, and Mbed OS (version 5) [3], the open-source embedded operating system designed for IoT. There are several projects in FreeRTOS, each supports a different development board (or device). We cross-compile all the projects that support the ARM architecture. For the Mbed OS, we compile all the targets that support the ARM architecture.
- (2) **Type-II: Linux Kernel** Linux kernel has been used on ARM devices widely. Such devices include mobile devices and ARM servers. For mobile devices, we use the most popular operating system Android, and build the kernel (version 4.4.169) for Android 9.0 (code name: Pie). We also build the kernel for Debian (version 9.6.0), one of the most popular Linux distributions for desktop computers and servers. We download and cross-compile the kernel (version 4.9.144) from Debian's official repository.
- (3) **Type-III: User-level Programs** We also use user-level programs, including daemons, libraries used on mobile devices, desktop computers and servers. Specifically, we build user-level programs from Buildroot [6], Android Open Source Project (AOSP, version 9.0.8) [1], and the popular Debian packages. They are representative programs for low-end embedded systems, mobile devices, and ARM desktop/servers. In particular, Buildroot is commonly used in low-end embedded systems, including routers, IP cameras and etc. We compile all binaries targeting ARM development boards. For Debian packages, we use the top five mostly installed packages, i.e., `libpam-modules`, `libattr1`, `libpam0g`, `zlib1g`, and `ebianutils`, ranked by the Debian popularity contest [9].

Table 2 shows the result of compiling options for Type-I, Type-II and Type-III binaries. We find that the Thumb instruction set is mostly used in Type-I binaries, and 02 and 0s are commonly used optimization levels. Due to this observation, we compile the benchmark programs for the evaluation to both ARM and Thumb instruction sets with 02 and 0s optimization levels to reflect real situations of ARM binaries in the wild.

**Obfuscation** To evaluate the impact of obfuscation, we use the O-LLVM [43], an open-source obfuscator, to compile the SPEC CPU2006. O-LLVM supports three different obfuscation methods, i.e., instruction substitution(*sub*), bogus control flow graph (*bcf*) and control flow flattening (*fla*). We apply each obfuscation method to each program, and then combine three methods together. Since the bogus control flow graph (*bcf*) consumes too much time (more than 2 hours) when applying it to C++ programs, we do not apply this method to C++ programs.

**Table 2: The compiling options for Type-I, Type-II and Type-III binaries. The third column shows the number of total object files (.o files), and the last two columns show the number of object files with the Thumb instruction set and optimization levels.**

	Name	# of Objects	Boards/Targets	Thumb	Optimization Levels
Type-I	Mbed	39,183	61	39,183	{'0s': 39,183}
	FreeRTOS	87	9	22	{'0s': 24, '02': 32}
Type-II	Linux Kernel (Android)	1,361	1	0	{'0s': 1, '02': 1, 291, '00': 1}
	Linux Kernel (Debian)	1,860	1	0	{'03': 1, '02': 1, 788, '00': 1, '0s': 1}
Type-III	AOSP	3,384	1	2,875	{'0s': 2, 787, '02': 299, '00': 27, '03': 69}
	Buildroot	188,387	103	1,677	{'0s': 188,387}
	Debian Packages	339	5	0	{'02': 305, '03': 34}

**Summary:** In total, we get 1,896 binaries including 248 obfuscated ones. We believe this dataset is representative to demonstrate the diversity of compilers, compiling options, target architectures and types of devices.

### 3.2 Determine Disassembly Primitives

In this work, we consider the instruction and function boundary as fundamental primitives (Table 1). Other ones (e.g., direct control flow graph) could be built upon them.

**Instruction Boundary** The instruction boundary refers to the start offset of an instruction, as well as the correct instruction set (ARM or Thumb). The purpose is two-fold. First, it is used to distinguish between code and inline data. Inline data is commonly used in ARM binaries, e.g., for the PC-relative addressing. This is different from x86 binaries, which do not contain inline data [25] except for the jump tables of binaries compiled by Visual Studio. Second, it is used to distinguish between ARM and Thumb instruction sets. This is challenging since the instruction set is partially determined by the target address of an (indirect) branch instruction, which is hard to be obtained by static analysis.

**Function Boundary** The function boundary refers to the start offset of a function. Function boundary recognition is a necessary primitive to construct the call graph, which is critical to the whole program analysis.

Interestingly, tools that could precisely recover instruction and function boundary do not scale. For instance, certain tools cannot finish the analysis within two CPU hours for some binaries. That means these tools are not scalable to large and complex binaries. Table 3 shows the number of binaries (the Timeout column) that cannot be analyzed within two CPU hours.

### 3.3 Generate Ground Truth

After determining the primitives, we need to get the ground truth. However, even with debugging symbols, it is not straightforward to directly get the result. we describe our approaches as follows.

**Instruction Boundary** We use mapping symbols [2] in the binaries to get the information of the instruction boundaries. Mapping symbols are generated by compilers to identify inline transitions between code and data, as well as ARM and Thumb instruction sets. There are three types of mapping symbols, including:

- \$a: Start of a region of code containing ARM instructions.
- \$t: Start of a region of code containing Thumb instructions.
- \$d: Start of a region of data.

For instance, the mapping symbol “0001043c \$t” denotes that the offset 0x0001043c in the binary is code (not inline data), with the Thumb instruction set.

However, mapping symbols only include the start address of the code and data regions without indicating the offset and the instruction set of each instruction in the region. To deal with this issue, we use Capstone [7] to retrieve the offset of each instruction. It works well since we have the instruction set (ARM/Thumb) information of each code region to help Capstone disassemble the code region.

Note that, the mapping symbol is an architecture-specific extension of the ARM ELF file. It may not exist in other architectures. By leveraging it, our system can detect the instruction boundary with a sound and complete result. Previous work can only detect 98% of the ground truth and requires a manual verification [25].

**Function Boundary** We leverage DWARF [37], a debugging file format to retrieve the function boundary. DWARF uses the data structure named Debugging Information Entry (DIE) to describe each variable, type, and function, etc. Each DIE has a tag (i.e., DW\_TAG\_subprogram) for function and each function has a key (i.e., DW\_AT\_low\_pc) to represent the function start address.

We extract the DW\_TAG\_subprogram and DW\_AT\_low\_pc from the DWARF of each binary to get the ground truth.

### 3.4 Extract the Result

We evaluate eight state-of-the-art ARM disassembly tools, including five noncommercial ones, i.e., angr [58], BAP [30], Objdump [18], Ghidra [11], Radare2 [22], and three commercial ones, i.e., Binary Ninja [5], Hopper [12] and IDA Pro [13]. Each tool has different ways to extract the instruction and function boundary. We carefully read the manual of each tool and write a script to extract the result.

## 4 EVALUATION

As discussed in Section 3, we build 1,896 binaries (including 248 obfuscated ones) to evaluate eight disassembly tools. We address the following research questions in Section 4.2, Section 4.3, Section 4.4 and Section 4.5 respectively.

- **RQ1:** What is the accuracy of disassembly tools towards the whole data set?
- **RQ2:** What are the factors that affect the results of disassembly tools, and what are the reasons?
- **RQ3:** Do different types and options of tools have different results?
- **RQ4:** How efficient are these disassembly tools?

**Table 3: The result of whole data set.** Invalid means the tool cannot identify any instructions or functions. Timeout means the tool cannot finish the analysis in two hours (CPU time). Exception means the tool raises exceptions during the analysis. Segfault means the tool triggers a segment fault during the analysis.

Tool Type	Tool	Instruction Boundary				Function Boundary				# of Timeout	# of Exception	# of Segfault
		Precision	Recall	F1 Score	# of Invalid	Precision	Recall	F1 Score	# of Invalid			
Noncommercial	angr	0.886	0.797	0.830	1	0.404	0.667	0.490	1	16	<b>364</b>	<b>262</b>
	BAP	0.565	0.277	0.309	<b>11</b>	0.533	0.358	0.387	<b>11</b>	<b>214</b>	0	0
	Objdump	0.702	0.750	0.722	0	-	-	-	-	0	0	0
	Ghidra	0.954	0.828	0.873	0	0.855	0.714	0.766	0	13	0	0
	Radare2	0.749	0.837	0.788	0	0.906	0.432	0.521	0	7	22	0
	Binary Ninja	0.984	0.857	0.900	0	0.806	0.800	0.781	0	37	0	0
Commercial	Hopper	0.971	<b>0.986</b>	<b>0.978</b>	0	0.825	<b>0.816</b>	0.807	0	2	0	0
	IDA Pro	<b>0.994</b>	0.970	<b>0.978</b>	5	<b>0.944</b>	0.781	<b>0.838</b>	5	1	0	0

## 4.1 Evaluation Metrics

We use *precision* and *recall* to measure the accuracy (or effectiveness) of a tool. The definition of these two metrics is in equation 1.

$$\text{precision} = \frac{tp}{tp + fp} \quad \text{recall} = \frac{tp}{tp + fn} \quad (1)$$

In the equation, we use *tp*, *fp*, *fn* to denote true positives, false positives and false negatives. *Recall* measures the ratio of true positives to the ground truth. A disassembler with high false negatives may have low recall. *Precision* measures the ratio of true positives to the result of a tool. A disassembler with high false positives may have low precision.

Considering the importance of both recall and precision, we also compute the F1 score according to equation 2. F1 score can reflect the overall accuracy of a tool.

$$F1 \text{ score} = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \quad (2)$$

## 4.2 Overall Results (RQ1)

Table 3 shows the overall result. The recall, precision and F1 score are computed in the granularity of macro-averaging. A tool may not be able to detect any instruction or function for a given binary; We mark such cases with the flag *Invalid*. We also set a threshold (two CPU hours in our study) for each tool to analyze a binary. This is because if a tool cannot finish the analysis in two hours, then it is not scaled to analyze a large number of binaries. We count the number of binaries that cannot be analyzed in two CPU hours with the flag *Timeout*. We also count the number of binaries that trigger an exception or a segment fault for each tool. We mark them with the flag *Exception* and *Segfault*, respectively.

Note that, a tool may have different options when performing the analysis. For instance, *angr* provides an option to disable or enable the resolution of indirect jumps. We use the default option for each tool to calculate the overall result and leave the evaluation of the impact of different options in Section 4.4.2.

**Instruction Boundary** *IDA Pro* has the highest precision value, while *Hopper* owns the highest recall value. Both of them have the highest F1 scores and are commercial tools. Moreover, these two are robust, since they do not raise any exceptions or generate any segment faults during the analysis. Among all the tools, *BAP* does not perform very well on both the instruction boundary and the function boundary. This is due to the insufficient support of the Thumb instruction set. Besides, *BAP* does not disassemble instructions that are out of the range of recognized functions. That means

if a function cannot be detected, then all instructions inside that function will be ignored. This is the reason why the recall of the instruction boundary is rather low. For other tools, the reason for the lower precision and recall mainly comes from two different reasons. One is the challenges raised by mixed ARM and Thumb instruction sets, and the other is the inline data.

**Function Boundary** *IDA Pro* still has the highest precision while *Hopper* has the highest recall. In terms of F1 score, *IDA Pro* has the highest value. It means that the function boundary is correlated with the instruction boundary. *BAP* mainly uses signatures learnt from a set of binaries to detect the function boundary. Due to the imprecise function signatures, functions with no representative signatures cannot be detected by *BAP*. As for *Radare2*, the recall is relatively low compared with other tools. That is because *Radare2* has a very strict policy on detecting functions. Users can use the command `aaaa` to explore more functions by searching for the function patterns.

**Robustness and Scalability** We find some noncommercial tools are not robust. For instance, more than 600 binaries triggered either an exception or a segment fault of *angr*. For the 262 binaries that triggered a segment fault, 160 of them are binaries compiled from the SPEC CPU2006, and 87.5% (140/160) of them are compiled using the Thumb instruction set. Based on this observation, there is a great space for *angr* to improve the support of the Thumb instruction set. This observation also applies to other tools, e.g., *Radare2*.

*BAP* does not scale well because 214 binaries cannot be analyzed in two hours, which is far more than other tools. We also observed timeouts when evaluating other tools except *Objdump*. For example, 37 binaries cannot be analyzed by *Binary Ninja* in two hours.

**Failed Cases** Disassembly tools failed to identify the right instruction boundary or function boundary due to different kinds of reasons. We manually study the failed cases and find that tools utilizing function signatures to identify the function boundary can make mistakes due to insufficient function signatures. Apart from this, disassembly tools cannot identify the accurate instruction boundary due to the inline data and mixed instruction set. We illustrate three different types of failed cases in the following.

Figure 4(a) shows an example of the false positive of *BAP*. There is a function starting from the offset 0x9cf0e4, but *BAP* thinks the function starts from the offset 0x9cf0e8. That is because the function signature used by *BAP* is not precise enough. Since ARM binaries vary due to different compilers and compiling options, it is challenging to timely update function signatures. Figure 4(b) shows an example of the false negative. *BAP* could locate the instruction at

Ground Truth function start 0009fe4 push {r4, lr} BAP function start 0009fe8 ldr r3, [r0, #0x8] 0009fec ldr lr, =#0x80080000 0009cff0 ldr ip, [r0, #0x8] 0009cff4 bics r2, lr, r3 0009cff8 mov r4, r0 0009ffc ldr lr, [ip] 0009d00 beq loc_9d040	00077650 b1 sub_e0008 00077654 sub s1, s1, #0x1 00077658 str r0, [sp, #0x58] 0007765c mov r3, #0x2 00077660 mov r2, #0x4
(a) An example of false positive	(b) An example of false negative

Figure 4: False positive and false negative of BAP

000710b4 movs r0, r5 000710b6 bl sub_70e8c 000710ba b sub_70fb8+98	}	Thumb ✓
sub_710bc: Indirect jumped		
000710bc ldrb r4, [r4, r2, lsr #22] 000710c0 ldrb r4, [r2, r2, lsr #22] 000710c4 ldrb r4, [r0, r2, lsr #22] 000710c8 strb r4, [lr, r2, lsr #22]	}	ARM ✗

Figure 5: Hopper misidentifies the instruction set.

d94a0: 0410a0e1 mov r1, r4 d94a4: 34309de5 ldr r3, [sp, #0x34] d94a8: c05a9fe5 ldr r5, [0x000d9f70] d94ac: 00609de5 ldr r6, [sp]	
d9f60: 08209de5 ldr r2, [sp, #8] d9f64: 102083e5 str r2, [r3, #0x10] d9f68: 4cd08de2 add sp, sp, #0x4c d9f6c: f08fbde8 pop {r4, r5, r6, r7, r8, sb, sl, fp, pc} d9f70: 78a46ad7 invalid d9f74: 56b7c7e8 stm r7, {r1, r2, r4, r6, r8, sb, sl, ip, sp, pc} d9f78: db702024 strhhs r7, [r0], -0xdb d9f7c: eecabd1 invalid d9f80: af0f7cf5 invalid ...	

Figure 6: Radare2 cannot identify the inline data. It tries to disassemble data as code, even the disassembled instructions are invalid.

the offset 0x77650 and disassembles it using the right instruction set. There is a function call instruction at offset 0x77650 and the callee function is starting from the 0xee008. However, BAP cannot identify the function (0xee008). This is because the prologue of the function (0xee008) does not satisfy the signature used by BAP.

Figure 5 shows an example, where Hopper uses a wrong instruction set to disassemble the binary. The instruction set from the offset 0x710bc is Thumb. However, Hopper disassembles it using the ARM instruction set. We further locate the potential root cause of this error. Specifically, the basic block (0x710bc) is indirectly reached from other basic blocks, thus it is hard for the tool to determine the right instruction set. Remember that, the instruction set is determined by the last bit of the target address.

Figure 6 shows a failed case that is caused by inline data. For instance, Radare2 disassembles the inline data (starting from the offset 0xd9f70), although it is an invalid instruction. In fact, the detection of inline data could be solved by a data reference analysis. Specifically, if we find that the offset 0xd9f70 has been referred by a load instruction at the offset 0xd94a8, then the offset 0xd9f70 is inline data with high confidence, instead of code.

### 4.3 Different Factors (RQ2)

Our data set consists of binaries that are built using different compilers, compiling options, and target architectures. Some of them are

ARM: 00070078 b1 sub_9eaa0 (Ghidra thinks function 0x9eaa0 will return) 0007007c ldr r3, [sp, #0x78 + var_44] 00070080 tst r3, #0x1 00070084 beq loc_70184
Thumb: 000505bc b1 sub_70668 (Ghidra thinks function 0x70668 is a non-return function) (Ghidra thinks the offset from 0x505c0 is inline data) 000505c0 movs r3, #0x1 000505c2 ldr r2, [sp, #0x80 + var_44] 000505c4 tst r3, r2

Figure 7: Ghidra performs differently when instruction set is different

even obfuscated. They represent the diversity of existing binaries in the wild. In the following, we further explore multiple factors that affect the accuracy of disassembly tools.

**4.3.1 Instruction Sets.** ARM and Thumb instruction sets are widely used in real-world binaries. To evaluate the impact of instruction sets, we divide binaries into two categories. The first one contains binaries compiled with the flag `-mthumb`, which use the Thumb instruction set. We call them Thumb set binaries. The other one is compiled *without* the flag `-mthumb`. By default, compilers use ARM instruction set. We call them ARM set binaries.

Figure 8 shows the evaluation result. The solid line and dotted line in the figure are used to denote the precision and recall, respectively. The x-axis shows the name of tools and the y-axis represents the average value of recall and precision for all the binaries. Note that, this format also applies to Figures 9, 10, 11, 12, 13 and 14.

First, disassembly tools perform worse for Thumb set binaries, i.e., they have lower precision and recall for the instruction boundary and the function boundary. Specifically, BAP has very low recall (0.40) and precision (0.01) for Thumb set binaries. We verified and reported our findings to developers of BAP. They acknowledged that BAP cannot handle Thumb binaries. Tools like Objdump cannot handle Thumb binaries either. This is because Objdump uses the ARM instruction set to linearly disassemble a binary without switching the instruction set. There are significant differences between the two instruction sets for tools like angr and Ghidra. This is because these tools have much better support of the ARM instruction set than the Thumb one.

Second, even for the binaries compiled from the same source code, the Thumb instruction set makes an inconsistency between the result. That is because the instruction set may have side effects on the recognized property of identified functions. Figure 7 shows such an example. The instructions at the offset 0x00070078 and 0x000505bc are same (BL), which represent a function call. Both function calls refer to the same callee according to the source code. However, since the instruction set is different, Ghidra misinterprets that the callee function in the Thumb instruction set is a non-return function, thus it completely ignores the code after that offset (0x000505bc). Several similar cases are observed for the tool. This is the reason why the recall is relatively low for Thumb set binaries of Ghidra.

Third, the result of the function boundary correlates with the instruction boundary. That's because these two primitives have a strong connection with each other. If the instruction boundary

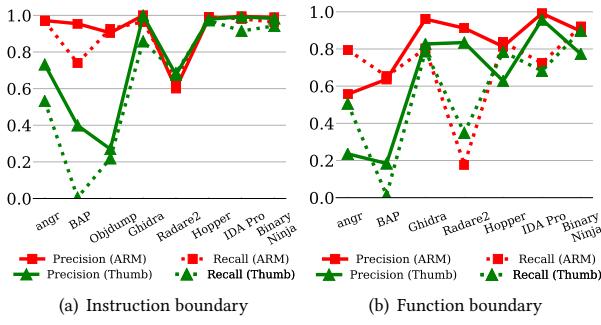


Figure 8: The result of different instruction sets

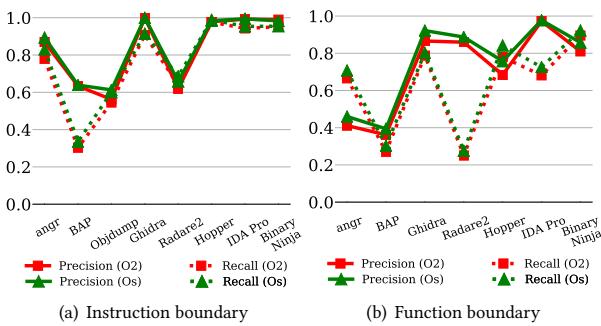


Figure 9: The result of different optimization levels

cannot be recognized precisely, it will greatly affect the recognition of the function boundary, and vice versa.

Fourth, the result of the function boundary is worse for the Thumb instruction set. We suspect that is due to the reuse of the BL `labe1` instruction as both a function call and a direct branch for Thumb set binaries. Specifically, the BL `labe1` (BLX `labe1`) instructions are used to directly invoke a function. For the ARM instruction set, compilers use instructions, e.g., B `labe1` for a direct branch. However, for the Thumb instruction set, the range of the B `labe1` is limited ( $\pm 2KB$  for 16-bit Thumb) [4]. Compilers tend to reuse the BL `labe1` for a direct branch (range is  $\pm 4MB$  for 16-bit Thumb), which is same with a function call. This confuses the disassembly tools, which misinterpret direct branches as function calls. This raises high false positives to identify the function boundary and results in a low precision. Due to this, the proposed method to identify function boundary without relying on function signatures in Nucleus [36] is also ineffective, since it assumes the function call instruction could be identified. The initial result of applying this tool to binaries with the Thumb instruction set show that both the precision and recall are below 0.12.

**Summary:** The Thumb instruction set does bring serious challenges to disassembly tools.

**4.3.2 Optimization Levels.** As shown in the Section 3.1, optimization levels `O2` and `Os` are mostly used ones. They represent the optimization for performance and size, respectively. To evaluate

Table 4: F1 scores for different optimization flags with at least 95% probability value. NA: not applicable.

Tool	angr	BAP	Objdump	Ghidra	Radare2	Hopper	IDA Pro	Binary Ninja
Instruction	0	0	0	0.005	0.065	0.014	0.001	0.018
Function	0.033	0.046	NA	0.020	0.037	0.053	0.033	0.037

the impact of optimization levels, we divide binaries into two categories. One contains binaries compiled with the `O2` flag, while the other one contains binaries compiled with the `Os` flag.

Figure 9 shows the result. Surprisingly, there is no significant differences between these two flags in terms of both recall and precision. To verify the conclusion that optimization level does not bring significant difference, we conducted an extra hypothesis test. We compute the F1 scores of the binaries in the two categories and compute the differences between every pair of binaries (i.e. one compiled with the flag `O2` while the other one compiled with the flag `Os`). We then randomly picked 40 samples and conducted t-test on the samples. Table 4 shows the result. We noticed that different optimization levels do not bring significant differences on all the eight tools. The maximum differences in terms of F1 score is only 0.065.

We further explore the potential reason. It turns out that the `Os` flag enables all the optimization methods introduced in the `O2` flag. Besides, it includes the ones to reduce binary size [8, 10], e.g., reducing the padding size and alignment. These ones have little impacts for the disassembly tool to identify the instruction and function boundary.

**Summary:** Optimization levels (`O2` and `Os`) do not bring significant differences.

**4.3.3 Compilers.** GCC and Clang are two popular compilers. To evaluate the impact of compilers, we build binaries (the SPEC CPU2006) with both GCC and Clang. Figure 10 shows the evaluation result.

For the instruction boundary, most tools do not have obvious differences between binaries built with different compilers except BAP, which will be explained later. However, for the function boundary, Radare2 and BAP are sensitive to binaries built with different compilers. For Radare2, the precision of the function boundary for binaries built with GCC is higher than the binaries built with Clang. BAP has a higher precision of the function boundary for binaries compiled with GCC. That is because BAP has a better collection of function signatures for binaries compiled with GCC than the ones compiled with Clang. Remember that, BAP does not disassemble the instructions that are not in the detected functions. Thus, the precision of the instruction boundary will also be higher for binaries compiled with GCC.

**Summary:** Compilers do not affect most of the tools, except Radare2 and BAP, mainly due to the function identification method used by them.

**4.3.4 Target CPU Architectures.** ARM has multiple architectures, e.g., ARMv7 and ARMv5. Each architecture has different hardware features. For instance, the 16-bit Thumb instruction (Thumb-1) is available from ARMv4, while the 32-bit Thumb-2 instructions

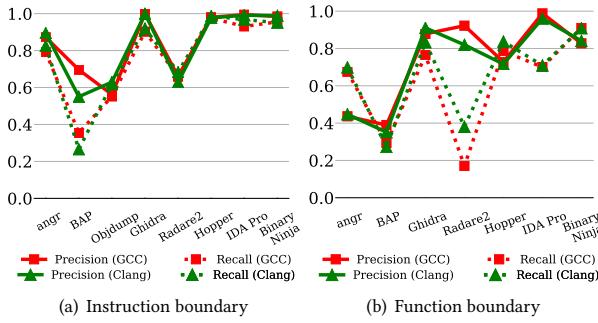


Figure 10: The result of different compilers

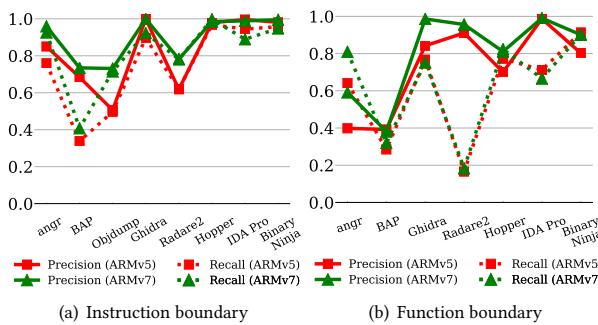


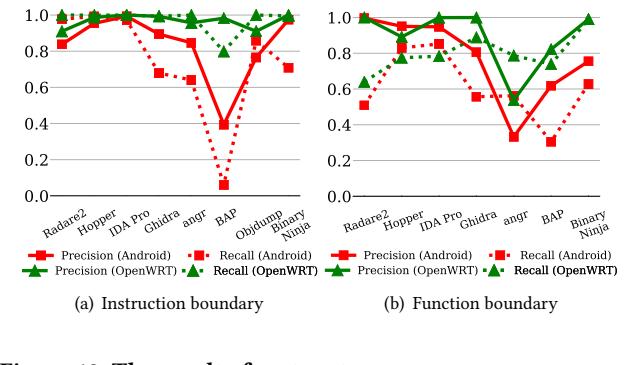
Figure 11: The result of different CPU architectures

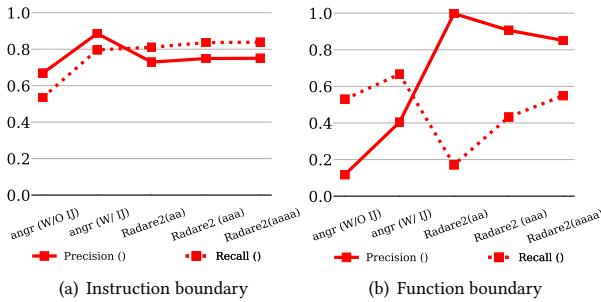
are available from ARMv6. Thus, if the binary is built for different architectures, instructions generated by the compilers will be different.

To evaluate the impact of binaries built with different CPU architectures, we use the binaries compiled for ARMv7 (`march=armv7-a`) and AVRv5 (`march=armv5t`). Figure 11 shows the result. We find that disassembly tools perform better for binaries with the ARMv7 architecture, in terms of the precision of function boundary. This is because the Thumb-2 instructions are supported in the ARMv7 architecture, where the `B label` instruction has a much larger jump range ( $\pm 16\text{MB}$ ) than the original one ( $\pm 2\text{KB}$  in the Thumb-1 instruction set) [4]. Compilers tend to use the `B label` instruction for the direct branch, instead of reusing the `BL label` instruction that is usually for the direct function call (Section 4.3.1). Thus, disassembly tools can distinguish the function call instruction with the direct branch instruction, and identify the function boundary more precisely.

**Summary:** For the ARMv7 CPU architecture, compilers use `B label` instruction for a direct branch, instead of reusing the `BL label` instruction. This helps the disassembly tools distinguish the direct branch instruction with the function call instruction, leading to a better precision value of identifying the function boundary.

**4.3.5 System Types.** ARM binaries exist in different types of systems. In our work, we also evaluate the impact of different types of binaries. In particular, we use the binaries built from the OpenWRT [19] (Linux based embedded systems used for routers, IP





**Figure 14: The Result of tools' options. W/ and W/O IJ means the indirect jump resolving is enabled and disabled for angr.**

We observe that obfuscation does not affect the instruction boundary too much. However, the function boundary is greatly affected by the control flow flattening. This is because the control flow flattening generates a huge number of fake control flows. These fake control flows are using the `BL` label instructions in the Thumb binaries for direct branches. These instructions confuse the disassembly tools and introduce false positives to the function boundary (Section 4.3.1).

**Summary:** Obfuscation introduces challenges to the disassembly tools to locate the function boundary, especially the control flow flattening. The root cause is due to the reuse of `BL` label instruction for direct branches, which are inserted by the obfuscation tool.

#### 4.4 Types and Options of Tools (RQ3)

**4.4.1 Commercial vs Noncommercial Tools.** In our work, we use eight state-of-the-art tools. Among them, there are three commercial tools, i.e., IDA Pro, Binary Ninja and Hopper, and five noncommercial ones. We find that commercial tools have higher precision and recall. As shown in Table 3, for the instruction boundary, the three commercial ones are ranked as top three in terms of both precision and recall. For the function boundary, these commercial tools are performing better than other ones, except that Radare2 has the better precision. Moreover, the commercial tools are more stable and robust. They do not trigger any segment faults or exceptions during the analysis.

**Summary:** Compared with noncommercial ones, commercial tools are more accurate, robust, and stable.

**4.4.2 Disassembly Tools' Options.** Disassembly tools have different options, which can affect the result. We use angr and Radare2 as examples since they provide explicit options that could be changed during the analysis. Figure 14 shows the result. Specifically, angr provides an option to enable or disable the indirect jump resolving. We observe that enabling the indirect jump resolving will increase the precision and recall, since it can resolve more code sections that could only be reached through indirect branches.

As for Radare2, it provides three different options. They are `aa,aaa` (the default value) and `aaaa`. Option `aa` only analyzes the function symbols, while option `aaa` adopts more analysis methods, including function calls, type matching analysis, value pointers. Option `aaaa` uses the function preludes to locate more functions and

performs constraint type analysis, besides the analysis included in the option `aaa`. We find that, complex analysis does not increase the accuracy of the instruction boundary, but has impacts on the function boundary. That is because the option `aa` only detects functions based on symbols, thus it misses most functions in the stripped binaries that do not have symbols. Options `aaa` and `aaaa` adopt more analysis methods, e.g., function preludes analysis, that greatly improve the result.

**Summary:** Disassembly tools' options affect the result. For angr, enabling indirect jump resolving can improve the result, while Radare2 has a better result for function boundary when using the option `aaaa`.

#### 4.5 Efficiency of the Tools (RQ4)

Efficiency is an important feature of disassembly tools. Efficient tools can handle large binaries within a reasonable time. We report the efficiency of the tools. In particular, we calculate the CPU usage, CPU times and memory consumption during the analysis. Our experiments are done in Ubuntu 18.04 with 128GB memory size and 30 core intel(R) Xeon(R) Silver 4110 CPUs. Specifically, we use the Python library psutil [21] to extract the related information about resource consumption, and then use the function `cpu_times` to obtain the CPU times. We also use the function `cpu_percent` (`interval = 1`) to extract the CPU percentage. Note that this value can be bigger than 100% in case of a process running multiple threads. We use the function `memory_info` to obtain the memory consumption. The memory size is the Resident Set Size (rss), which is the non-swapped physical memory a process has used. The result is shown in Figure 15.

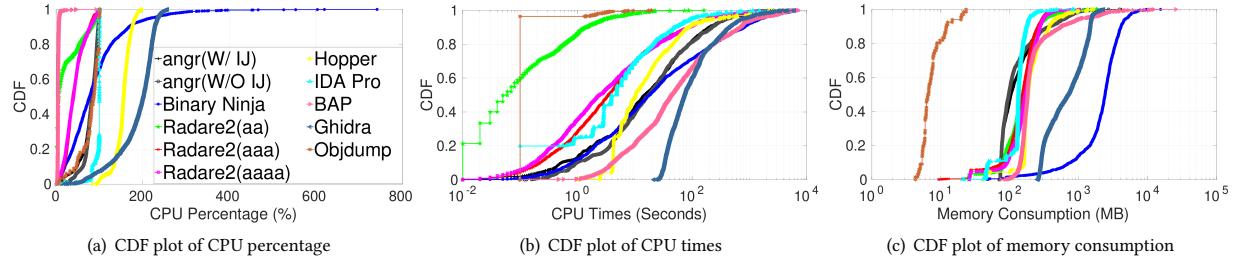
**CPU Percentage** Binary Ninja consumes lots of CPU resource and can reach to nearly 800% for some binaries. Ghidra ranks the second. Half of the binaries would consume 200% of the CPU usage. BAP consumes the least CPU percentage. However, according to our observation, BAP spends a lot of time to analyze the binaries due to the inefficient usage of CPU.

**CPU Times** Among all the tools, Ghidra consumes most CPU times compared with the other tools in nearly 80% binaries. There are no significant performance differences for angr with and without indirect jump resolving. Since the indirect jump resolving improves the result, we recommend users to enable this option during analysis. For Radare2, the option `aaa` needs much more CPU times compared with the option `aa`. However, the precision and recall of the instruction boundary do not have a significance improvement.

**Memory Consumption** Among all the tools, Binary Ninja consumes most of the memory (nearly 1 GB in maximum), while Objdump consumes the least. IDA Pro is quite stable for the memory usage. It consumes only around 100MB memory for nearly 70% of the binaries.

#### 4.6 Threats to Validity

Threats to validity mainly lies in the data set we choose. To reduce this threat, we tried our best to make our data set representative, we surveyed more than 200,000 objects in Section 3.1 to get the most popular compiling options. We use the surveyed compiling options to compile the benchmark programs to get the binaries.



**Figure 15: The evaluation result of performance. The legend in the latter two figures are same with the first one.**

Considering the impact of obfuscation, we also use the state-of-the-art obfuscators (i.e. O-LLVM) to compile the programs into obfuscated binaries with different obfuscated mechanisms. Apart from the binaries compiled from benchmark programs, we also considered the programs in the real world. We use binaries in AOSP and OpenWRT, which can represent the binaries for mobile systems and IoT devices.

## 5 IMPLICATIONS

In this section, we discuss implications based on the evaluation result and point out possible improvements.

**ARM-specific disassembly strategies** First, inline data is popular in ARM binaries. Previous research shows that there is few inline data in x86/x64 binaries and the jump tables are located in the `.rodata` section. However, inline data is very common in ARM binaries, which increases the difficulty to locate instruction boundaries. Second, there are two instruction sets, i.e., ARM and Thumb instruction sets. Detecting the right instruction set is challenging for disassembly tools. Furthermore, we noticed that most tools do not have good support on the Thumb instruction set, either with a wrongly detected instruction set or a thrown exception. For instance, `angr` throws exceptions and gets segment faults for several binaries with the Thumb instruction set. `Objdump` can merely identify the Thumb instruction set. Given the fact that the Thumb instruction set is popular, especially in the binaries for mobile systems, there is an urgent need to propose effective solutions. Third, since most existing works are focusing on x86 and x64 [25, 27, 28, 36, 61, 62], some ARM specific mechanisms should be proposed to deal with the instruction set switching. For instance, the hybrid disassembly technique [68] could be leveraged to locate the inline data and distinguish between different instruction sets, with customizations to adapt to the ARM architecture. Besides this, disassembly tools could perform a further check on its disassembly result. In other words, they could conduct a conflict analysis to improve the result. For example, `Radare2` explicitly knows when there is an invalid instruction. In this case, it can either switch the mode, or further check whether the invalid code is actually inline data through a data reference analysis.

**Mechanisms to identify the function boundary** Our result shows that there is still a large space to improve the effectiveness of detecting the function boundary. Tools usually use function signatures to identify functions. These signatures could be generated

through a machine learning based method. However, the machine-learning based methods could be limited due to the incompleteness of the training data sets [36]. For instance, the machine-learning based method in `BAP` performs worse than most of the other tools in detecting function boundaries. Furthermore, the mechanisms that work well on x86/x64 [36] cannot be applied to ARM, because ARM does not have a distinguished function call instruction, which is required by the method. According to our evaluation, besides function call, `BL label` is widely used in the Thumb instruction set for direct branch. Disassembly tools cannot distinguish the usage of `BL label` as direct branch with direct function call, resulting in a low precision in terms of the function boundary.

We think a more effective algorithm to detect the function boundary is needed. For example, developers could use the machine-learning based mechanism to detect the function first, and then conduct a static analysis by considering the internal logic between different basic blocks to reduce the false positives and false negatives. Moreover, disassembly tools can further analyze the `BL label` instruction to understand whether it's a function call. We think a further analysis of the usage of the `BL label` instruction can greatly improve the result of the function boundary.

**Usability** Tools have different user interfaces and plugin infrastructures. For instance, `IDA Pro`, `Hopper` and `Binary Ninja` have user-friendly GUI interfaces, and provide easy-to-use Python APIs. `angr` itself does not provide GUI, and is invoked purely through a Python script. `BAP` has a good flexible architecture for extension. However, the supported language Ocaml has a steep learning curve, compared to the Python programming language. As for `Radare2`, it is completely different from the other tools. It just loads the binary and provides an interactive shell. Users have to leverage the shell to perform the analysis. There are many different kinds of built-in analysis phases.

We also observe that non-commercial tools suffer from the scalability and stability. For instance, `BAP` cannot finish the analysis on several binaries while `angr` will raise exceptions or get segment faults on several binaries. Furthermore, tools may have different options, which impact the usage of system resources. Users should pick the right options according to the purpose. For example, if users use the `Radare2` to disassemble the instruction (and do not care about the function boundary), they can use the option `aa`, which satisfies the need and is much faster than other options.

## 6 DISCUSSION

First, with the introduction of the ARMv8 architecture, there exist 64-bit ARM binaries, which are missed in this work. However, 32-bit binaries are still the most popular ones. Due to the compatibility concern, new ARM architectures maintain backward compatibility with old ones. Our findings in this paper can still be applied to ARMv8 (ARMv8 supports both AArch64 for 64-bit binaries and AArch32 for 32-bit binaries) and future versions of ARM as long as ARM does not deprecate 32-bit ARM instruction set. Besides, AArch64 simplifies the task of disassembly tools. This is because 32-bit ARM has both 16-bit and 32-bit instructions and much more diverse branch instructions. As shown in our evaluation, the switching between instruction sets brings serious challenges to disassembly tools.

Second, we only evaluate eight state-of-the-art disassembly tools. However, there exist some disassemblers that are either research prototypes or not actively maintained. They are excluded from our work. Moreover, we only evaluate two fundamental disassembly primitives. We think other primitives such as direct control flow graph or direct call graph are easy to be generated if the instruction boundary and function boundary are located correctly<sup>2</sup>.

Third, the generation of the ground truth is an essential step. Fortunately, the ARM ELF format introduces mapping symbols that can help to distinguish between different instruction sets, and between code and inline data. By leveraging this information, we can generate a complete and sound result for the instruction boundary. At the same time, we could use the DWARF debugging information to extract the ground truth of the function boundary. For other primitives like control flow graph and call graph, they consist of direct jumps and indirect jumps. Direct jumps can be built based on the precise instruction boundary and function boundary, which is the reason why we do not include them in the evaluation. For the evaluation of indirect jumps, we cannot get a sound and complete ground truth even if we have the source code. This is because some jump targets can only be determined at running time. We leave the evaluation of these primitives as one of the further works.

## 7 RELATED WORK

The most related one is the study of x86/x64 disassembly tools [25]. However, there are several differences between the study and ours. First, ARM supports mapping symbols, which can help us to extract the precise ground truth. Second, we focus on the metrics of recall and precision when evaluating the different factors (e.g. optimization levels and instruction modes), which can reveal the tools' limitations. Third, the conclusion of x86/x64 study cannot be applied to ARM binaries. For example, inline data is very common in ARM binaries, which is rare in x86/x64 binaries. Apart from this, there are two instruction set in ARM architecture, which require more precise analysis mechanism for tools to identify the right instruction set. Furthermore, the reuse of BL 1label instruction for both function call and direct branch brings challenges to disassemblers to detect the function boundary.

Zhang et al. [68] combined the linear sweep and recursive traversal. However, their work is for x86 binaries and there is no experiment describing the accuracy of this algorithm. Ben et al. [29] proposed the idea of speculative disassembly on Thumb binaries

<sup>2</sup>We understand that the indirect control flow transfer is still a challenging task.

with the assumption that binaries are not obfuscated. However, their work is not scalable to real world binaries as ARM instruction set are widely used. Though Kruegel et al. [45] proposed an algorithm on obfuscated binaries, their work does not target the ARM architecture. Bauman et al. [28] proposed the idea of superset disassembly, while Miller et al. [49] proposed the probabilistic disassembly mechanism. However, they only focus on x86/x64 binaries.

Detecting the function boundary is also a challenging research topic. Rosenblum et al. [54] use the machine learning technique to identify functions. Other works [27, 30, 42, 44, 56] extended this idea with different machine learning algorithms. However, it is rather hard to build a general model. Other tools [20, 46, 58] use heuristics or hard-coded signatures to identify the function boundary. The fundamental problem is that there exist functions that do not have the signatures or do not follow the heuristics. Qiao et al. [53] applied static analysis to detect the function boundary. However, their work only targets the x86 architecture. Dennis et al. [36] designed a new methodology on detecting function boundaries by analyzing control flow graphs. However, it assumes there is a distinguished function call instruction (e.g., the call instruction in x86/x64) in the binary. This mechanism cannot be applied to ARM due to no distinguished function call instruction in ARM binaries.

As discussed, disassembly tools have been widely used in existing works. Taegyu et al. [60] designed and implemented a rewriter for ARM binaries. It requires precise disassembly results. Vulnerability detection by applying binary similarity techniques [48, 64, 70] relies on the accurate disassembly results. Different algorithms [50, 66, 68, 69] have been proposed to build control flow graphs. Since some applications are leveraging off-the-shelf tools to disassemble binaries, the evaluation can help improve these tools, which further improves the result of applications that depend on these tools.

## 8 CONCLUSION

In this paper, we conduct the first comprehensive study on the capability of eight ARM disassembly tools to locate instruction and function boundaries, using diverse ARM binaries built with different compiling options and compilers. We report our new findings, which shed light on the limitations of the state-of-the-art disassembly tools, and point out potential directions for improvements.

## ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Zhi Wang for the helpful discussion. This work is supported in part by Hong Kong RGC Project (No. 152239/18E), the National Natural Science Foundation of China (NSFC) under Grant 61872438, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), Zhejiang Key R&D 2019C03133, Singapore National Research Foundation, under its National Cybersecurity R&D Program (Grant Nos.: NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Grant Nos.: NRF2018NCR-NSOE003-0001), NRF Investigatorship (Grant Nos.: NRFI06-2020-0022), and DARPA under agreement number FA875019C0003.

## REFERENCES

- [1] Android Open Source Project. <https://source.android.com/>.
- [2] ARM Mapping Symbols. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0474f/CHDGFCDL.html>.
- [3] Arm Mbed OS. <https://www.mbed.com/en/>.
- [4] B, BL, BX, BLX, and BXJ. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489c/Cihfdaf.html>.
- [5] Binary Ninja: A New Kind Of Reversing Platform. <https://binary.ninja/>.
- [6] Buildroot: Making Embedded Linux Easy. <https://buildroot.org>.
- [7] Capstone: The Ultimate Disassembly. <http://www.capstone-engine.org/>.
- [8] Clang: Documentation. <https://clang.llvm.org/docs/CommandGuide/clang.html>.
- [9] Debian Popularity Contest. [https://popcon.debian.org/by\\_inst](https://popcon.debian.org/by_inst).
- [10] GCC: Options That Control Optimization. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.
- [11] Ghidra: A Software Reverse Engineering(SRE) Suite of Tools Developed by NSA. <https://ghidra-sre.org/>.
- [12] Hopper Disassembler. <https://www.hopperapp.com/>.
- [13] IDA Pro. <https://www.hex-rays.com/products/ida/>.
- [14] Issues submitted to BAP. <https://github.com/BinaryAnalysisPlatform/bap/issues/951>.
- [15] Issues submitted to Binary Ninja. <https://github.com/Vector35/binaryninja-api/issues/1359>.
- [16] Issues submitted to Ghidra. <https://github.com/NationalSecurityAgency/ghidra/issues/657>.
- [17] Issues submitted to Radare2. <https://github.com/radareorg/radare2/issues/14223>.
- [18] Objdump - Display Information from Object Files. <https://linux.die.net/man/1/objdump>.
- [19] OpenWRT. <https://openwrt.org/>.
- [20] Pardyn Project. Dyninst: Putting the Performance in High Performance Computing. <https://www.dyninst.org/>.
- [21] Psutil. <https://psutil.readthedocs.io>.
- [22] Radare2. <https://rada.re/r/>.
- [23] The FreeRTOS Kernel. <https://www.freertos.org/>.
- [24] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 23th ACM Conference on Computer and Communications Security*.
- [25] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-depth Analysis of Disassembly on Full-scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium*.
- [26] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium*.
- [27] Tiffany Bao, Johnathan Burkett, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23th USENIX Conference on Security Symposium*.
- [28] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, Ahmad M Mustafa, Gbadebo Ayoade, Khaled Al-Naami, Latifur Khan, Kevin W Hamlen, Bhavani M Thuraisingham, Frederico Araujo, et al. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Network and Distributed Systems Security Symposium*.
- [29] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative Disassembly of Binary Code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*.
- [30] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*.
- [31] Cristina Cifuentes and Mike Van Emmerik. 2001. Recovery of jump table case statements from binary code. *Science of Computer Programming* 40, 2-3 (2001), 171–188.
- [32] Andrei Costin, Jonas Zaddach, Aurelien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium*.
- [33] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*.
- [34] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23rd Symposium on Network and Distributed System Security*.
- [35] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [36] Andriesse Dennis, Asia Slowinska, and Bos Herbert. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*.
- [37] Michael J. Eager. Introduction to the DWARF Debugging Format. <http://www.dwarfstd.org/doc/DebuggingusingDWARF-2012.pdf>.
- [38] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Network and Distributed System Security Symposium*.
- [39] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium*.
- [40] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 23th ACM Conference on Computer and Communications Security*.
- [41] Grant Hernandez, Farhaan Fowze, Tuba Yavuz, Kevin RB Butler, et al. 2017. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*.
- [42] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. 2011. Labeling Library Functions in Stripped Binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*.
- [43] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LVMM – Software Protection for the Masses. In *Proceedings of the 1st International Workshop on Software Protection*.
- [44] Nikos Karampatziakis. 2010. Static Analysis of Binary Executables Using Structural SVMs. In *Proceedings of the 23rd Advances in Neural Information Processing Systems*.
- [45] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium*.
- [46] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Proceedings of the 12th USENIX Security Symposium*.
- [47] Tamás László and Ákos Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30, 1 (2009), 3–19.
- [48] Chandramohan Mahinthan, Xue Yinxing, Xu Zhengzi, Liu Yang, Cho Chia Yuan, and Tan Hee Beng Kuan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.
- [49] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*.
- [50] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained Control-flow Integrity Through Binary Hardening. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.
- [51] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE.
- [52] Manish Prasad and Tzi-cker Chiueh. 2003. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks.. In *Proceedings of the USENIX Annual Technical Conference*.
- [53] Rui Qiao and R Sekar. 2017. Function interface analysis: A principled approach for function recognition in COTS binaries. In *Proceedings of the 47th International Conference on Dependable Systems and Networks*.
- [54] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code.. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*.
- [55] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of Executable Code Revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering*.
- [56] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks.. In *Proceedings of the 24th USENIX Conference on Security Symposium*.
- [57] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice: Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 22th Annual Symposium on Network and Distributed System Security*.
- [58] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok(State of) the Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*.
- [59] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. 2018. BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid. In *Proceedings of the 27th USENIX Security Symposium*.

- [60] Kim Taegyu, Chung Hwan Kim, Choi Hongjun, Yonghwi Kwon, Brendan Saltafornaggio, Xiangyu Zhang, and Dongyan Xu less. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proceedings of the 37th Annual Computer Security Applications Conference*.
- [61] Veen Victor, Goktas Enes, Contag Moritz, Pawlowski Andre, Chen Xi, Rawat Sanjay, Bos Herbert, Holz Thorsten, Athanasopoulos Elias, and Giuffrida Cristiano. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*.
- [62] Ruoyu Wang, Yan Shoshtaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security*.
- [63] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*.
- [64] Xue Yinxing, Xu Zhengzi, Chandramohan Mahinthan, and Liu Yang. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149.
- [65] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Symposium on Network and Distributed System Security*.
- [66] Chao Zhang, Tao Wei, Zhao Feng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE.
- [67] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 25–36.
- [68] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*.
- [69] Mingwei Zhang and R Sekar. 2015. Control Flow and Code Integrity for COTS Binaries: An Effective Defense Against Real-world ROP Attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*.
- [70] Xu Zhengzi, Chen Bihuan, Chandramohan Mahinthan, Liu Yang, and Song Fu. 2017. Spain: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*.



# How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools using Bug Injection

Asem Ghaleb

aghaleb@alumni.ubc.ca

University of British Columbia

Vancouver, Canada

Karthik Pattabiraman

karthikp@ece.ubc.ca

University of British Columbia

Vancouver, Canada

## ABSTRACT

Security attacks targeting smart contracts have been on the rise, which have led to financial loss and erosion of trust. Therefore, it is important to enable developers to discover security vulnerabilities in smart contracts before deployment. A number of static analysis tools have been developed for finding security bugs in smart contracts. However, despite the numerous bug-finding tools, there is no systematic approach to evaluate the proposed tools and gauge their effectiveness. This paper proposes *SolidiFI*, an automated and systematic approach for evaluating smart contracts' static analysis tools. *SolidiFI* is based on injecting bugs (i.e., code defects) into all potential locations in a smart contract to introduce targeted security vulnerabilities. *SolidiFI* then checks the generated buggy contract using the static analysis tools, and identifies the bugs that the tools are unable to detect (false-negatives) along with identifying the bugs reported as false-positives. *SolidiFI* is used to evaluate six widely-used static analysis tools, namely, Oyente, Securify, Mythril, SmartCheck, Manticore and Slither, using a set of 50 contracts injected by 9369 distinct bugs. It finds several instances of bugs that are not detected by the evaluated tools despite their claims of being able to detect such bugs, and all the tools report many false positives.

## CCS CONCEPTS

- Security and privacy → Software and application security;

## KEYWORDS

Ethereum, Ethereum security, solidity code analysis, smart contracts, smart contracts security, smart contracts analysis, smart contracts dataset, static analysis tools evaluation, bug injection, fault injection

### ACM Reference Format:

Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools using Bug Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397385>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397385>

## 1 INTRODUCTION

The past few years have witnessed a dramatic rise in the popularity of smart contracts [19]. Smart contracts are small programs written into blocks running on top of a blockchain that can receive and execute transactions autonomously without trusted third parties [28]. Ethereum [17] is the most popular framework for executing smart contracts.

Like all software, smart contracts may contain bugs. Unfortunately, bugs in smart contracts can be exploited by malicious attackers for financial gains. In addition, transactions on Ethereum are immutable and cannot be reverted, so losses cannot be recovered. Further, it is difficult to update a smart contract after its deployment. Consequently, there have been many bugs in smart contracts that have been maliciously exploited in the recent past [1, 3, 34].

Several approaches and tools have been developed that statically find security bugs in smart contracts [26, 33, 36, 42, 43]. However, despite the prevalence of these static analysis tools, security bugs abound in smart contracts [39]. This calls into question the efficacy of these tools and their associated techniques. Unfortunately, many of the static analysis tools have been evaluated either only by their developers on custom data-sets and inputs, often in an ad-hoc manner, or on data-sets of contracts with a limited number of bugs (112 bugs [24] and 10 bugs [37]). *To the best of our knowledge, there is no systematic method to evaluate static analysis tools for smart contracts regarding their effectiveness in finding security bugs.*

Typically, static analysis tools can have both false-positives and false-negatives. While false positives are important, false negatives in smart contracts can lead to critical consequences, as exploiting bugs in contracts usually leads to loss of ether (money). Also, empirical studies of software defects in the field have found that many of the defects can be detected by static analysis tools in theory, but are not detected due to limitations of the tools [41]. In our work, we focus mostly on the undetected bugs (i.e., false negatives), though we also study false-positives of the tools.

We perform bug injection to evaluate the false-negatives of smart contract static analysis tools. Bug injection as a testing approach has been extensively explored in the domain of traditional programs [15, 23, 40]; however, there have been few papers on bug injection in the context of smart contracts. This problem is challenging for two reasons. First, smart contracts on Ethereum are written using the Solidity language, which differs from conventional programming languages typically targeted by mutation testing tools [21]. Second, because our goal is to inject security bugs, the bugs injected should lead to exploitable vulnerabilities.

This paper proposes *SolidiFI*<sup>1</sup>, a methodology for systematic evaluation of smart contracts' static analysis tools to discover potential flaws in the tools that lead to undetected security bugs. *SolidiFI* injects bugs formulated as code snippets into *all* possible locations into a smart contract's source code written in Solidity. The code snippets are vulnerable to specific security vulnerabilities that can be exploited by an attacker. The resulting buggy smart contracts are then analyzed using the static analysis tools being evaluated, and the results are inspected for those injected bugs that are not detected by each tool - these are the false-negatives of the tool. Because our methodology is agnostic of the tool being evaluated, it can be applied to any static analysis tool that works on Solidity.

We make the following contributions in this paper.

- Design a systematic approach for evaluating false-negatives and false-positives of smart contracts' static analysis tools.
- Implement our approach as an automated tool, *SolidiFI*, to inject security bugs into smart contracts written in Solidity.
- Use *SolidiFI* to evaluate six static analysis tools of Ethereum smart contracts for false-negatives and false-positives.
- Provide an analysis of the undetected security bugs and false-positives for the 6 tools, and the reasons behind them.

The results of using *SolidiFI* on 50 contracts show that all of the evaluated tools had significant false-negatives ranging from 129 to 4137 undetected bugs across 7 different bug types despite their claims of being able to detect such bugs, as well as many false positives. Further, many of the undetected bugs were found to be exploitable when the contract is executed on the blockchain. Finally, we find that *SolidiFI* takes less than 1 minute to inject bugs into a smart contract (on average). Our results can be used by tool developers to enhance the evaluated tools, and by researchers proposing new bug-finding tools for smart contracts.

## 2 BACKGROUND

### 2.1 Smart Contracts

As mentioned earlier, smart contracts are written in a high-level language such as Solidity. They are compiled to Ethereum Virtual Machine (EVM) bytecode that is deployed and stored in the blockchain accounts. Smart contract transactions are executed by miners, which are a network of mutually untrusted nodes, and governed by the consensus protocol of the blockchain. Miners receive execution fees, called gas, for running the transactions which are paid by the users who submit the execution requests. We illustrate smart contracts through a running example shown in Figure 1 (adapted from prior work [13]).

This contract implements a public game that enables users to play a game and submit their guesses or solutions for the game along with some amount of money. The money will be transferred to the account of the last winner if the guess is wrong; otherwise the user will be set as the current winner and will receive the money from users who play later. The *constructor()* at line 6 runs only once when the contract is created, and it sets the initial winner to the owner of the contract defined by the user who submitted the create transaction of the contract (*msg.sender*). It also initializes the *startTime* variable to the current timestamp during the contract

```

1 pragma solidity >=0.4.21 <0.6.0;
2 contract EGame{
3     address payable private winner;
4     uint startTime;
5
6     constructor() public{
7         winner = msg.sender;
8         startTime = block.timestamp;
9     }
10    function play(bytes32 guess) public {
11        if(keccak256(abi.encode(guess)) == keccak256(abi.
12            encode('solution'))){
13            if (startTime + (5 * 1 days) == block.
14                timestamp){
15                winner = msg.sender;
16            }
17        }
18    }
19    function getReward() payable public{
20        winner.transfer(msg.value);
21    }
22 }
```

Figure 1: Simple contract written in Solidity.

creation. The function *play* at line 10 is called by the user who wants to submit his/her guess, and it compares the received guess with the true guess value. If the comparison is successful, it sets the *winner* to the address of the user account who called this function, provided the guess was submitted within 5 days of creating the contract. Finally, the function *getReward* sends the amount of *ether* specified in the call to *getReward* (*msg.value*), to the last winner.

### 2.2 Static Analysis Tools

We consider six static analysis tools for finding bugs in smart contracts in this paper, Oyente, Securify, Mythril, Smartcheck, Manticore, and Slither. They all operate on smart contracts written in Solidity, and are freely available. Further, they are all automated and require no annotations from the programmer. We selected Oyente [33], Securify [43], Mythril [36], and Manticore [35] as they were used in many smart contract analysis studies [16, 37, 39, 43]. We included Smartcheck [42] as it uses a pattern matching approach rather than symbolic execution employed by the previous four tools. Similarly Slither [26] is another non-symbolic-execution based tool, but unlike SmartCheck, it uses Static Single Assignment (SSA) for analysis.

## 3 MOTIVATION AND CHALLENGES

This section first presents motivating examples of undetected security bugs by static analysis tools, followed by an overview of the challenges in the evaluation of the tools.

### 3.1 Motivating Examples

The contract example in Figure 1 has at least 2 vulnerabilities, (1) two instances of timestamp dependency bug at lines 8 and 12, and (2) one instance of transaction ordering dependence (TOD) represented by the transactions at lines 13 and 16. The timestamp dependency bug is that the block's timestamp should not be used in the transaction, while the TOD bug is that the state of the smart contract should not be relied upon by the developer (Section 7).

<sup>1</sup>*SolidiFI* stands for Solidity Fault Injector, pronounced as Solidify.

```

11 uint _vtime = block.timestamp;
12 if (startTime + (5 * 1 days) == _vtime){

```

**Figure 2: Modification made to the contract in Figure 1**

We have used four of the static analysis tools in Section 2 (supposed to detect these bugs) to check this contract for bugs, Oyente, Securify, Mythril, and SmartCheck. According to the tools' research papers [33, 36, 42, 43], Oyente, Mythril and SmartCheck should detect the timestamp dependency bug. However, we found that while Oyente and Mythril were not able to detect both instances of timestamp dependency bug in lines 8 and 12, Smartcheck detected only the instance in line 12. For the second instance, Smartcheck gave a hint that `block.timestamp` should be "used only in equalities". To further test SmartCheck's ability to detect the bug, we made a small modification to the syntax of the smart contract while keeping its semantics the same (Figure 2). SmartCheck subsequently failed to detect the bug altogether.

Regarding the TOD bug, both Oyente and Securify are supposed to detect this class of bugs. However, we found that only Securify detected this bug successfully while Oyente was not able to detect it. We extracted the code snippet representing TOD from this contract (lines 10 to 16), injected it in another larger contract free of bugs, and obtained similar results.

These examples motivated us to prepare multiple code snippets for the different bugs (within the scope of the tools) and to manually inject them into the code of 5 smart contracts (the first 5 contracts in the set of contracts in Section 7). We then used the tools to check the buggy contracts, and found several instances of undetected bugs even though the tools were supposed to detect them. However, it was tedious and error-prone to manually inject these bugs and inspect the results, and so we decided to automate this process.

### 3.2 Automated Bug Injection Challenges

The simplest way to inject bugs into smart contracts is to inject them at random locations - this is how traditional fault injection (i.e., mutation testing) works. However, random injection is not a cost-effective approach as we have to follow specific guidelines for the injected bug to be exploitable. We identify two main challenges.

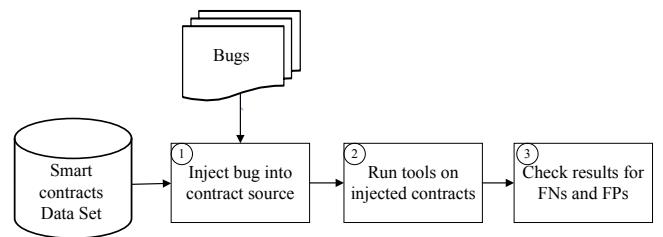
**3.2.1 Bug Injection Locations.** As the underlying techniques used by some tools (e.g., symbolic execution) depends on the control and data flow in the analyzed contracts, injecting an instance of each bug at a single location would not be sufficient. Therefore, bugs should be injected into all potential locations in the contract code. On the other hand, the process of identifying the potential locations depends on the code of the original contract, and also on the type and nature of each bug. Injecting bugs at the wrong locations would result in compilation errors. In addition, it might yield instances of dead code in the contract. For example, injecting a bug formulated as a stand-alone function inside the body of another function would result in a compilation error, as Solidity does not support nested functions. Moreover, a bug injected into an 'if' statement condition that would make the condition always fail, would make the 'then' clause unreachable.

**3.2.2 Semantics Dependency.** For the injected bug to be an active bug that can be exploited by an attacker, it has to be aligned with the semantics of the original contract. For example, assume that we want to inject a Denial of Service (DoS) bug by calling an external contract. We can use an if-statement with a condition containing a call to another contract function. However, for this bug to be executed, we also need to define the appropriate external contract.

*SolidiFI* addresses the first challenge by parsing the Solidity language into an Abstract Syntax Tree (AST) and injecting bugs into *all* syntactically valid locations. It addresses the second challenge by formulating exploitable code snippets for each bug type.

## 4 SOLIDI FI APPROACH AND WORKFLOW

The main goal of *SolidiFI* is to perform systematic evaluation of *static analysis* tools used to check smart contracts for known security bugs. Figure 3 shows the workflow of *SolidiFI*. The code snippets representing a specific security bug are injected in each smart contract's source code at *all* possible locations (step 1). The selection of the injection locations is a function of the bug to be injected. *SolidiFI* injects bugs into the source code to imitate the introduction of bugs by developers. However, its use is not restricted to tools that perform analysis at the source code level. For example, tools that work on the EVM bytecode would compile the buggy contracts to produce the EVM code for analysis. Then, the injected code is scanned using the static analysis tools (step 2). Finally, the results of each tool are checked, and false negatives and false positives are measured (step 3).



**Figure 3: SolidiFI Workflow.**

### 4.1 Bug Model

In our work, a security bug is expressed as a code snippet, which leads to a vulnerability that the security tool being analyzed aims to detect. *SolidiFI* reads code snippets to be injected from a pre-defined bug pool prepared by us (the bug pool can be easily extended by users to add new bugs). For each tool, we only inject the bugs that the tool claims to detect, based on the tool's research paper. However, because the tools are continuously evolving, the research paper may not have the up-to-date list of bugs detected by the tool, and hence we use the tool's online documentation to augment it.

### 4.2 Bug Injection

In this work, the security bugs are injected in the source code in three ways as follows.

**4.2.1 Full Code Snippet.** In this approach, we prepare several code snippets for each bug under study. Each code snippet is a piece of code that introduces the security bug.

To illustrate the process, we discuss the bugs and example code snippets.

**Timestamp Dependency.** The current timestamp of the block can be used by contracts to trigger some time-dependent events. Given the decentralized nature of Ethereum, miners can change the timestamp (to some extent). Malicious miners can use this capability and change the timestamp to favor themselves. This bug was exploited in the GoverMental Ponzi scheme attack [2]. Therefore, developers should not rely on the precision of the block's timestamp. Figure 4 shows an example of a code snippet that represents the bug (`block.timestamp` returns the block's timestamp).

```
1 function bug_tmstmp() public returns(bool)
2 {   return block.timestamp >= 1546300; }
```

Figure 4: Timestamp dependency examples.

**Unhandled Exceptions.** In Ethereum, contracts can call each other, and send *ether* to each other (e.g., `send` instruction, `call` instruction, etc.). If an exception is thrown by the callee contract (e.g., limited gas for execution), the contract is terminated, its state is reverted, and `false` is returned to the caller contract. Therefore, unchecked returned values within the caller contract could be used to attack the contract, leading to undesired behavior. A serious version of this bug occurred in the “King of the Ether” [2]. Figure 5 shows an example (the `send()` instruction requires its return value to be checked for exceptions to make it secure).

```
1 function unhandledSend() public {
2     callee.send(5 ether); }
```

Figure 5: Unhandled exceptions examples.

**Integer Overflow/Underflow.** In Solidity, storing a value in an integer variable bigger or smaller than its limits lead to integer overflow or underflow. This can be used by attackers to fraudulently siphon off funds. For example, Figure 6 shows an example code snippet in which an attacker can reset the `lockTime` for a user by calling the function `incrLockTime` and passing 256 as an argument – this would cause an overflow, and end up setting the lockTime to 0. Batch Transfer Overflow is a real-world example [7].

```
1 function incrLockTime(uint _sec) public{
2     lockTime[msg.sender] += _sec; }
```

Figure 6: Integer overflow/underflow example.

**Use of `tx.origin`.** In a chain of calls, when contracts call functions of each other, the use of `tx.origin` (that returns the first caller that originally sent the call) for authentication instead of `msg.sender` (that returns the immediate caller) can lead to phishing-like attacks [10]. Figure 7 shows an example snippet in which `tx.origin` is used to withdraw money.

```
1 function bug_txorigin(address _recipient) public {
2     require(tx.origin == owner);
3     _recipient.transfer(this.balance); }
```

Figure 7: `tx.origin` authentication example.

**Re-entrancy.** Contracts expose external calls in their interface. These external calls can be hijacked by attackers to call a function within the contract itself several times, thereby performing unexpected operations within the contract itself. For example, the external call in Line 3 of the snippet code shown in Figure 8 can be used by an attacker to call the `bug_reEntrancy()` function repeatedly, potentially leading to withdrawal of ether more than the balance of the user. The DAO attack [1] is a well-known example exploiting this bug.

```
1 function bug_reEntry(uint256 _Amt) public {
2     require(balances[msg.sender] >= _Amt);
3     require(msg.sender.call.value(_Amt));
4     balances[msg.sender] -= _Amt; }
```

Figure 8: Re-entrancy example.

**Unchecked Send.** Unauthorized Ether transfer, such as non zero sends, can be called by external users if they are visible to public, even if they do not have the correct credentials. This means unauthorized users can call such functions and transfer ether from the vulnerable contract [10]. An example code snippet is shown in Figure 9.

```
1 function bug_unchkSend() payable public{
2     msg.sender.transfer(1 ether); }
```

Figure 9: Unchecked send example.

**Transaction Ordering Dependence (TOD).** Changing the order of the transactions in a single block that has multiple calls to the contract, results in changing the final output [13]. Malicious miners can benefit from this. An example code snippet vulnerable to this bug is shown in Figure 10. In this example, the attackers can send a puzzle solving reward to themselves instead of the winner of the game by executing `bug_tod2()` before `bug_tod1()`.

```

1 address payable winner_tod;
2 function setWinner_tod() public {
3     winner_tod = msg.sender;
4 function getReward_tod() payable public{
5     winner_tod.transfer(msg.value);

```

**Figure 10:** TOD example.

**4.2.2 Code Transformation.** This approach aims to transform a piece of code without changing its functionality, but make it vulnerable to a specific bug. We leverage known patterns of vulnerable code to inject this bug. We use this approach to inject two bug classes that are compatible with this approach, namely (1) integer overflow/underflow and (2) use of tx.origin.

Table 1 shows examples of the code patterns that are replaced to introduce the bugs, and the vulnerable patterns for each bug type.

**Table 1:** Code transformation patterns.

Bug Type	Original Code Patterns	New Code Patterns
tx.origin	msg.sender==owner	tx.origin==owner
Overflow	bytes32	bytes8
Overflow	uint256	uint8

Figure 11 shows an example before and after bug injection using this approach. In this example, *transfer* instruction is used to perform a transfer of the specified ether amount to the receiver's account after verifying the direct caller of *sendto()* to be the owner. To inject the tx.origin bug, the authorization condition *msg.sender == owner* should be replaced with the *tx.origin == owner*, in which the owner is not the direct caller of *sendto()*. However, the authorization check is passed successfully, which enables attackers to authorize themselves, and send ether from the contract, even if they are not the owner.

```

1 /*(Before)*/
2 function sendto(address receiver, uint amount) public
3 {
4     require (msg.sender == owner);
5     receiver.transfer(amount);
6 /*(After injection)*/
7 function sendto(address receiver, uint amount) public
8 {
9     require (tx.origin == owner);
10    receiver.transfer(amount);

```

**Figure 11:** Code transformation example.

**4.2.3 Weakening Security Mechanisms.** In this approach, we weaken the security protection mechanisms in the smart contract code, which protect external calls. Note that our goal is to evaluate the static analysis tool, and not the smart contract itself. We use this approach to inject Unhandled exception bugs. Figure 12 shows an example, in which the *Unhandled exceptions* bug is injected by removing the *revert()* statement that reverts the state of the contract if the transfer transaction failed - this causes the balance to incorrectly become 0 even if the transaction failed.

```

1 /*(Before)*/
2 function withdrawBal () public{
3     Balances[msg.sender] = 0;
4     if(!msg.sender.send(Balances[msg.sender]))
5         { revert(); }}
6 /*(After injection)*/
7 function withdrawBal () public{
8     Balances[msg.sender] = 0;
9     if(!msg.sender.send(Balances[msg.sender]))
10    { //revert();
11 }

```

**Figure 12:** Weakening security example.

## 5 SOLIDIFI ALGORITHM

The process for injecting bugs takes as input the Abstract Syntax Tree (AST) of the smart contract, and has the following steps.

- (1) Identify the potential locations for injecting bugs and generate an annotated AST marking all identified locations.
- (2) Inject bugs into all marked locations to generate the buggy contract.
- (3) Check the buggy contract using the evaluated tools and inspect the results for undetected bugs and false alarms.

We discuss the steps in detail below.

**Bug Locations Identification:** The AST is passed to *Bug Locations Identifier*, that drives a bug injection profile (BIP) of all possible injection locations in the target contract for a given security bug. The BIP is derived using AST-based analysis for identifying potential injection locations in smart contract code by Algorithm 1. Algorithm 1 takes as input the AST and the bug type to be injected, and outputs the BIP.

---

### Algorithm 1 Identifying Injection Locations Algorithm

---

```

1: procedure FINDALLPOTENTIALLOCATIONS(AST, bugType)
2:   for Each form of code snippets in bugType do
3:     if snippetForm == simple statement then
4:       BIP ← WalkAST(simpleStatement)
5:     else if snippetForm == non-function block then
6:       BIP ← WalkAST(nonFunctionBlock)
7:     else if snippetForm == functionDefinition then
8:       BIP ← WalkAST(functionDefinition)
9:     end if
10:   end for
11:   BIP ← FindRelatedSecurityMechanisms
12:   BIP ← FindCodeThatCanBeTransformed
13:   return BIP
14: end procedure

```

---

To address the first challenge of identifying bug injection locations as mentioned in Section 3.2, we define rules that specify the relation between the bug to be injected and the target contract structure. In general, bugs take two forms: an individual statement, and a block of statements. A block of statements can be defined either as a stand-alone function, or a non-function block such as an 'if' statement. Therefore, we use a rule for each form of the bug that defines the specifications of the locations for injecting it.

To identify such locations, for each distinct form of the code snippets defining the bug type to be injected, we walk the AST

based on the code snippet form and the related rule (lines 2-10 in Algorithm 1). *WalkAST(simpleStatement)*, for example, will visit (parse) the AST and find all the locations where a simple statement can be injected without invalidating the compilation state of the contract, and the same for the other forms of the code snippets of the bug type. After identifying the locations for injecting code snippets of bugs, we also look for existing security mechanisms to be weakened to introduce the related bug, and the code patterns to be transformed for introducing the bug (lines 11 and 12).

**Bug Injection and Code Transformation:** *SolidiFI* uses a systematic approach to inject bugs into the potential locations in the target contract. The *Bug Injector* model seeds a bug for each location specified in the BIP. It uses text-based code transformation to modify the code where the information derived from the AST is used to modify the code to inject bugs. Three different approaches are used to inject bugs as discussed in Section 4.2. In addition to injecting bugs in the target contract, *Bug Injector* generates a *BugLog* that specifies a unique identifier for each injected bug, and the corresponding location(s) in the target contract where it has been injected.

**Buggy Code Check and Results Inspection:** The resulting buggy contract is passed to the *Tool Evaluator* that checks the buggy code using the tools under evaluation. It then scans the results generated by the tools looking for the bugs that were injected but undetected, with the help of *BugLog* generated by the *Bug Injector*. *SolidiFI* only considers the injected bugs that are undetected. So if an evaluated tool reported bugs in locations other than where bugs have been injected, *SolidiFI* does not consider them in its output of false negatives. This is to avoid potential vulnerabilities in the original contract from being reported by *SolidiFI*, which would skew the results. Moreover, *SolidiFI* inspects results generated by the tools looking for other reported bugs and checks if they are true bugs or false alarms (more details in Section 7).

## 6 IMPLEMENTATION

*SolidiFI* approach is fully-automated (except for the pre-prepared buggy snippets). This involves compiling the code, injecting and generating buggy contracts, running the evaluated tools on the buggy contracts, and inspecting reports of the evaluated tools for false-negatives, mis-identified cases, and false-positives (except for the manual validation of the filtered false-positives). To make *SolidiFI* reusable, we did not hard-code the patterns that are replaced to introduce bugs, but rather made them configurable from an external file. We have made *SolidiFI* code publicly available<sup>2</sup>.

*SolidiFI* uses the Solidity compiler **solc** (supports compiler versions up to 0.5.12) to compile the source code of the smart contract to make sure it is free from compilation errors before bugs are injected. In addition, *SolidiFI* uses **solc** to generate the AST of the original code in JavaScript Object Notation (JSON) format. We have implemented the other components of *SolidiFI* in Python in about 1500 lines of code. These components are responsible for identifying the potential locations for injection, injecting bugs using a suitable approach, generating the buggy contract, and inspecting the results of the evaluated tools and then reporting the undetected bugs and false alarms. Finally, we developed a Python client to interact with

contracts deployed on Ethereum network - this client is used for assessing the exploitability of the injected bugs in the generated buggy contracts.

## 7 EVALUATION

The aim of our evaluation is to measure the efficacy of *SolidiFI* in evaluating smart contract static analysis tools, and finding cases of undetected bugs (i.e., false negatives) and false positives. We also measure the performance of *SolidiFI* itself, as well as the ability to exploit the undetected bugs. We made all the experimental artifacts used in this study and our results publicly available<sup>3</sup>. Our evaluation experiments are thus derived to answer the following research questions:

- **RQ1.** *What are the false negatives of the tools being evaluated?*
- **RQ2.** *What are the false positives of the tools being evaluated?*
- **RQ3.** *Can the injected bugs in the contracts be activated (i.e., exploited) at runtime by an external attacker?*
- **RQ4.** *What is the performance of SolidiFI?*

As mentioned earlier, we have selected six static analysis tools for evaluation, Oyente [33], Securify [43], SmartCheck [42], Mytril [36], Manticore [35], and Slither [26]. We downloaded these tools from their respective online repositories, which are mentioned in the corresponding papers (Oyente 0.2.7, Securify v1.0 as downloaded and installed in Dec 2019, Mytril 0.21.20, Smartcheck 2.0, Manticore 0.3.2.1, and Slither 0.6.9).

To perform our experiments, we used a data set of fifty smart contracts, chosen from the list of verified smart contracts available on *Etherscan* [25], a public repository of smart contracts written in Solidity for Ethereum. We selected these contracts based on three factors namely (1) code size (we selected contracts with different sizes that were representative of Etherscan contracts ranging from small contracts with tens of lines of code to large contracts with hundreds of lines of code), (2) compatibility with Solidity version 0.5.12 (at the time of writing, 312 out of 500 verified smart contracts in EtherScan supported Solidity 0.5x and higher), and (3) contracts with a wide range of functionality (e.g., tokens, wallets, games). Table 2 shows the number of lines of code (including comments), and number of functions and function modifiers<sup>4</sup> for each contract. The contracts range from 39 to 741 lines of code (loc), with an average of 242 loc.

We limited ourselves to 50 contracts due to the time and effort needed to analyze the contracts by the evaluated tools and inspect the analysis results of the tools to verify false-positives. *With that said, even with this dataset, SolidiFI found significant numbers of undetected bugs in the tools (e.g., false negatives), as will be discussed in the following sections.*

As explained in Section 4, in our experiments, we injected bugs belonging to seven different bug types within the detection scope of the selected tools. Table 3 shows the bug types, and the tools that are designed to detect each bug type. We chose these bug types based on the bug types detected by the individual tools, and because these bugs are common in smart contracts, and lead to vulnerabilities that have been exploited in practice [1, 2, 6]. However, *SolidiFI* is not confined to these bug types.

<sup>3</sup><https://github.com/DependableSystemsLab/SolidiFI-benchmark>

<sup>4</sup>Function modifier checks a condition before the execution of the function.

<sup>2</sup><https://github.com/DependableSystemsLab/SolidiFI>

In our experiments, we set the time-out value for each tool to 15 minutes per smart contract and bug type. If a tool's execution exceeds this timeout value, we terminate it and consider the bugs found as its output. While 15 minutes may seem high, our goal is to give each tool as much leeway as possible. Only 2 of the tools exceeded this time limit in some cases (i.e., Mythril and Manticore). For these two tools, we experimented with larger timeout values, but they did not significantly increase their detection coverage. Note that the total time taken to run the experiments was already quite high with this timeout value - for example, it took us about 4 days to analyze the contracts using Mythril (50-contracts \* 6 bug-types \* 15-minutes = 75 hours).

**Table 2: Contracts benchmark. F represents Functions, and M represents Function Modifiers**

<b>Id</b>	<b>Lines</b>	<b>F+M</b>	<b>Id</b>	<b>Lines</b>	<b>F+M</b>	<b>Id</b>	<b>Lines</b>	<b>F+M</b>
1	103	6	18	406	29	35	317	29
2	128	9	19	218	32	36	383	20
3	132	10	20	308	27	37	368	24
4	117	6	21	353	18	38	195	24
5	250	17	22	383	19	39	52	4
6	161	22	23	308	20	40	465	22
7	165	22	24	741	27	41	160	8
8	251	17	25	196	12	42	128	16
9	249	19	26	143	20	43	285	22
10	39	5	27	336	33	44	298	24
11	193	19	28	195	24	45	156	14
12	281	27	29	312	13	46	125	6
13	161	8	30	711	57	47	223	18
14	185	20	31	216	12	48	232	19
15	160	8	32	143	14	49	52	4
16	248	27	33	129	16	50	171	18
17	128	17	34	445	29			
<b>Average values</b>				<b>242</b>	<b>18</b>			

**Table 3: Bug types used in our evaluation experiments:** \*\* means that the tool can detect the bug type.

<b>Bug Type</b>	<b>Oyente</b>	<b>Security</b>	<b>Mythril</b>	<b>SmartCheck</b>	<b>Manticore</b>	<b>Slither</b>
Re-entrancy	*	*	*	*	*	*
Timestamp dependency	*		*	*		*
Unchecked send		*	*			
Unhandled exceptions	*	*	*	*		*
TOD	*	*				
Integer overflow/underflow	*		*	*	*	
Use of tx.origin			*	*		*

## 7.1 RQ1: What Are the False Negatives of the Tools Being Evaluated?

The core part of our evaluation is to use *SolidiFI* to inject bugs, and evaluate the effectiveness of the tools in detecting the injected bugs. We performed the following steps in our experiments. First, *SolidiFI* is used to inject bugs of each bug type in the code of the 50 smart contracts, one bug type at a time. The resulting buggy contracts are then checked using the static analysis tools. Finally, the number of the injected bugs that were not detected by each tool were recorded.

To get meaningful results, we inject bugs that are as distinct as possible by preparing diverse set of distinct code snippets with different data inputs and function calls- this resulted in 9369 distinct bugs. We consider two injected bugs as distinct if the static analysis tool under study would reason about them differently based on the underlying methodology, where either the data and control flow leading to the injected bug is different, or the design patterns of the bug snippets are different. *To ensure a fair evaluation, we inject only the bugs that are supposed to be detected by each tool.*

We consider an injected bug as being correctly detected by a tool if and only if it identified both the line of code in which the bug was injected, as well as the bug type (e.g., Re-entrancy). In many cases, we observed that the tool would correctly identify the line of code in which the bug occurred, but would misidentify the bug type. Therefore, we also report the former separately .

The results of injecting bugs of each bug type, and testing them using the six tools are summarized in Table 4. In the table, “Injected bugs” column specifies the total number of injected bug for each bug type, “✓” means we did not find any undetected bug of that bug type (row), while “NA” means the bug type is out of scope of the tool, i.e., it is not designed to detect the bug type. The numbers for each column specify the total number of bugs that were either incorrectly detected or not detected by the tool corresponding to the bug type specified in that row. The number within parentheses specifies the number of cases that were not reported by the tool - this does not consider the incorrect reporting of the bug type.

From the table, we can see that a significant number of false negatives occur for all the evaluated tools, and that *none of the tools was able to detect all the injected bugs correctly even if we accepted a incorrect bug type with the correct line number as a detected bug.* In fact, the only tool that had 100% coverage for individual bug types was Slither, for Reentrancy and tx.origin bugs. Of all tools, Slither had the lowest false-negatives, followed by Securify across bug types.

Our results thus show that all static analysis tools have many corner cases of bugs that they are not able to detect. *Note that it is surprising that our technique found as many undetected bugs by the tools as it did, given that our goal was not specifically to exercise corner cases of the tools in question.* We will discuss the reasons for the missed detections and the implications later (Section 8).

## 7.2 RQ2: What Are the False Positives of the Tools Being Evaluated?

A false-positive occurs when a tool reports a bug, but there was no bug in reality. Unlike false negatives, where we know exactly where the bugs have been injected, and hence have ground truth, measuring false positives is challenging due to the lack of ground

**Table 4: False negatives for each tool. Numbers within parentheses are bugs with incorrect line numbers or unreported.**

Security bug	Injected bugs	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	1343 (844)	1008 (232)	232 (232)	1085 (805)	1343 (106)	1250 (1108)	✓
Timestamp dep	1381 (886)	1381 NA	810 (810)	902 (341)	NA	537 (1)	
Unchecked-send	1266	NA (449)	499 (389)	389 (389)	NA	NA	NA
Unhandled exp	1374 (918)	1052 (571)	673 (571)	756 (756)	1325 (1170)	NA	457 (128)
TOD	1336 (1199)	1199 (263)	263 NA	NA	NA	NA	NA
Integer flow	1333 (898)	898 NA	1069 (932)	1072 (1072)	1196 (1127)	NA	NA
tx.origin	1336	NA	NA (445)	445 (1120)	1239	NA	✓

truth. This is because we cannot assume that the smart contracts used are free of bugs (though they are chosen from the verified contracts on Etherscan). Further, manually inspecting each bug report and related contract involves a tremendous amount of effort due to the large number of bug reports, and is hence not practical.

To keep the problem of determining false-positives tractable, we came up with the following approach. The main idea is to manually examine only those bugs that are *not* reported by the majority of the other tools for each smart contract. In other words, we conservatively assume that a bug that is reported by a majority of the tools cannot be a false positive. However, at worst, we will underestimate the number of false positives in this approach, subject to the vagaries of the manual inspection process. We also verified that many of the bugs that are excluded by the majority are indeed false-positives by manually examining a random sample of them.

Even after this filtering, we had to manually inspect a significant number of bugs to determine if they were false positives. Therefore, for each tool, we randomly selected 20 bugs of each bug type category that were not excluded by the majority approach, and inspected them manually. For those cases where the number of bugs is less than or equal to 20, we inspected them all. Based on the results of our manual inspection, we estimated the false positives as the percentage of bugs inspected that were indeed false positives, multiplied by the number of bugs filtered (i.e., not excluded).

For example, assume that the total number of bugs reported by a tool is 100. Of these 100 bugs, let us assume that 60 are also reported by the majority of the other tools for the smart contract, and hence we exclude them. Of the remaining 40 filtered bugs, we manually examine 20 bugs chosen at random. Assume that 16 of these are indeed false-positives. We assume that 80% of the filtered bugs are false-positives, and estimate the number of false-positives to be 32.

The results of false positives reported by each tool are summarized in Table 5. In the table, the “Threshold” column refers to the majority threshold, which is the number of tools that must detect the bug in order for it to be excluded from consideration - this number depends on how many tools are able to detect the bug type.

For each tool, the sub-column “Reported” shows the number of bugs reported by the tool, the sub column “FL” shows the number of

bugs that have been filtered (not excluded) by the majority approach, while the sub column “FP” shows the false positives of the tool based on the manual inspection as explained above. Empty cells in the table represent cases where a tool was not designed to detect a bug type. Note that some of the tools detected bugs outside the 7 categories that we considered - we called these as *miscellaneous*.

From the table, we can see that all the evaluated tools have reported a number of false positives, ranging from 2 to 801 for most of the bug types. Interestingly, the results show that the tools with low numbers of false negatives reported high false positives, i.e., *Slither* and *Securify*. For example, although *Slither* was the only tool that successfully detected all the injected Re-entrancy bugs, it reported significant false positives. This raises the question of whether the high detection rate was simply a result of overzealously reporting bugs by the tool (this is also borne out by the high number of bugs reported under the *miscellaneous* category by this tool). *This highlights the need for security analysis tools that are able to detect bugs while maintaining low false positive rates.*

We provide some examples of the false-positive cases below. For example, most unhandled exception bugs were reported even though the code checks the return values of the send functions for exceptions using `require()`. As another example, many false positives were re-entrancy bugs where the code contains the required checks of the contract balance, and updates the contract states before the Ether transfer. Oyente reported several cases as integer over/underflow even though they are no integer related calculations (e.g., `string public symbol = "CRE";`). On the other side, we tried to be consistent with the assumptions considered by the tools during our manual inspection. For example, some of the cases that we considered as true bugs were re-entrancy bugs that use the `transfer` function. This function protects against re-entrancy issues as it has limited gas; however, we considered them as true bugs as the attack can happen if the gas price changes - this is detected by some of the tools (e.g., Slither).

### 7.3 RQ3: Can the Injected Bugs Be Activated by an External Attacker?

The goal of this RQ is to assess whether the undetected bugs in RQ1 can be activated in the contract at runtime. This is to determine whether the reason behind the bug not being detected by the evaluated tool was because the bug cannot be activated (and hence cannot be exploited by an attacker). We deploy the set of buggy contracts with the undetected bugs (found in RQ1) on the Ethereum blockchain, and execute transactions that attempt to activate them.

To conduct these experiments, we use MetaMask [9], a browser extension that allows us to connect to an Ethereum node called INFURA [8], and run our buggy contracts on this node. We have created Ethereum accounts on Ethereum Kovan Testnet (test network) using MetaMask, and deposited sufficient amount of Ether to these accounts to enable us to execute transactions (pay the required gas for transactions). We use Remix [4] (Solidity editor) to deploy contracts on Ethereum Kovan Testnet. Remix enables us to connect with MetaMask to deploy contracts on INFURA Ethereum node.

**Table 5: False positives reported by each tool. Empty cells mean that the tool was not designed for that particular bug type.**

Bug Type	Threshold	Oyente			Security			Mythril			SmartCheck			Manticore			Slither		
		Reported	FL	FP	Reported	FL	FP	Reported	FL	FP	Reported	FL	FP	Reported	FL	FP	Reported	FL	FP
=		0	0	-	12	12	12	54	54	43	0	0	-	6	6	6	79	79	71
Re-entrancy	4	0	0	-				12	12	0	0	0	-				12	12	0
Timestamp dep	3	0	0	-				7	4	4	14	3	3						
Unchecked send	2				7	4	4												
Unhandled exp	3	10	10	10	0	0	-	0	0	-	6	6	6				0	0	-
TOD	2	32	24	24	121	97	97												
Integer flow	3	947	943	801				17	3	3	3	2	2	9	9	9			
tx.origin	2							0	0	-	3	1	0				4	2	0
Miscellaneous		0			318			144			1520			169			1807		

We illustrate the process with an example. As mentioned in RQ1, Manticore did not report instances of injected integer overflow/underflow bugs - an example is shown in Figure 13. Our goal is to attempt to activate this bug in the deployed buggy contract by calling the function `bug_intou3()`. The returned result was 246 - this is not the expected value (-10) due to the use of unsigned integer type (i.e., `uint8`) instead of a signed integer type, which resulted in an integer underflow. Thus, the bug can be activated by an attacker.

We had to manually craft inputs for each bug in order to test its activation, which takes significant effort. Because of the large number of undetected bugs, we selected 5 undetected bugs for each bug type randomly from different contracts to test their activation. Table 6 shows the results of our activation experiments. In the table “-” means we were not able to perform experiments on this bug type, as it requires the attacker to behave as a miner, which would consume a significant amount of computational resources. The results show that one can exploit (activate) all the selected bugs in their related buggy contracts. Therefore, the infeasibility of activation of the bug was not the reason that the evaluated tools failed to detect the injected bugs.

```

1 function bug_intou3() public{
2     uint8 vundflw =0;
3     vundflw = vundflw -10; // underflow
4     return vundflw;

```

**Figure 13: Undetected integer underflow bug****Table 6: Activity of Selected Undetected Bugs.**

Bug type	Selected bugs	Activated bugs
Re-entrancy	5	5
Timestamp dependency	5	5
Unchecked send	5	5
Unhandled exceptions	5	5
TOD	-	-
Integer overflow/underflow	5	5
Use of tx.origin	5	5

## 7.4 RQ4: What Is the Performance of *SolidiFI*?

Finally, we measured the performance of *SolidiFI* in terms of the time it takes to inject bugs and generate buggy contracts. We excluded the time of running the tools being evaluated to check the buggy contracts, as this is tool-specific and independent of *SolidiFI*. We preformed injection of each bug type in each contract five times and calculated the average of the five runs, and then calculated the average of injecting the seven bug types in each contract. The average time of injecting all instances of bug types in a contract is 25 seconds, and the worst case time was 46 seconds (for contract 24, which was the largest contract in our set). *Thus, SolidiFI takes less than 1 minute on average per contract and bug type.*

## 8 DISCUSSION

In this section, we examine the reasons for the false negatives of the tools observed in RQ1. We then examine the implications of the results, and our methodology, on both tool developers and end users. Finally, we examine some of the limitations of *SolidiFI* and threats to validity of our experiments.

### 8.1 Reasons for False-Negatives

To establish a practical understanding of the presented results and why some bugs were not detected, we will highlight the code snippets for some of the bugs that were not detected, and then discuss the reasons behind them. We organize this discussion by tool.

**Oyente** was not able to detect many instances of injected reentrancy, timestamp dependency, unhandled exceptions, integer overflow/underflow, and TOD bugs as mentioned earlier<sup>5</sup>. According to the paper [33], Oyente works on detecting only re-entrancy bugs that are based on the use of `call.value`. Some of the recent tools, such as *Slither*, consider the detection of re-entrancy bugs with limited gas that are based on `send` and `transfer`. Those papers claim that `send` and `transfer` do not protect from re-entrancy bugs in case of gas price change. Furthermore, Oyente failed to detect instances of re-entrancy bugs that are based on the use of `call.value`. One of the TOD code snippets we used in our experiments is mentioned in the running example at Figure 1 on lines 9–16, which emulates a simple game and its winner. The malicious behavior

<sup>5</sup>Because the released version of Oyente did not work with the latest version of the Solidity compiler (0.5.12), we made few changes to the injected contracts to get it to work with Oyente - these did not impact the tool's coverage.

occurs when the two transactions are executed in one block and the attacker tries to change the order of the received transactions. To understand why this bug is not detected by Oyente, in Oyente the EVM bytecode is represented as a control flow graph (CFG). The execution jumps are used as edges that connect the nodes of the graph representing the basic execution blocks in the code. The symbolic execution engine of Oyente uses the CFG to produce a set of symbolic traces (execution paths), each associated with a path constraint and auxiliary data, to verify pre-defined properties (security bugs being detected). Basically, Oyente detects TOD by comparing the different execution paths and the corresponding data flow (Ether flow) for each path. Oyente reports those different execution paths that have different Ether flows.

For Oyente to be able to detect all TOD bugs successfully, the symbolic execution engine should generate all possible execution paths for the contract to find the erroneous path - this is challenging due to the incompleteness of symbolic execution. It also uses some bounds that limit the symbolic execution.

**Smartcheck** failed to detect most of the injected bugs across all the categories. Smartcheck checks for bugs by constructing an Intermediate Representation (IR) from the source code, and then using XPath patterns to search for bugs in the IR. This approach lacks accuracy as some bugs cannot be expressed as XPath expressions. For example, the re-entrancy bug is difficult to express as an XPath pattern, and is hence not detected.

Further, because Smartcheck uses XPath patterns that detect specific syntax of some bugs, even a slight variation in the syntax of the bug snippets would not match the XPath patterns. For instance, SmartCheck did not report some occurrences of unhandled exceptions. By checking the code snippet for one of the undetected unhandled exception bug depicted in Figure 14, we found that Smartcheck was not able to detect it as unchecked send because Smartcheck only looks for `send` functions without an if-statement (that checks the return value). However, in this snippet, the `send` is within an if-statement even though the `revert()` exists in the else clause of the if-statement, which will be triggered on the successes of `send`. The same happens when other functions are used for sending ether (e.g., `call`, etc.) instead of `send`. This is an inherent problem with syntactic, rule-based tools such as Smartcheck.

```
1 if (!addr.send (42 ether))
2   {receivers +=1;}
3 else
4   {revert();}}
```

Figure 14: Unhandled exception code snippet 1.

**Mythril** was the tool with the largest set of undetected bugs in our experiments. It failed to detect many instances of re-entrancy, timestamp dependency, unchecked send, unhandled exceptions, integer overflow/underflow and use of `tx.origin`. For example, the buggy code in Figure 15 was not detected by Mythril. The condition of the if-statement that checks the return value of the `send` will always be evaluated as true because of the added condition “`|| 1==1`”. Hence, the execution of the contract would be reverted in all cases by the function `revert()`, regardless of whether the `send` succeeds. *This is incorrect as the execution should only be reverted*

on the fails of `send`. However, Mythril does not detect this as it only evaluates the `send()` part in the condition of the if-statement rather than evaluating the whole condition with the OR part (`|| 1==1`).

Mythril is also very slow in term of the time it takes to analyze contracts. Although we set the time-out for analyzing each contract to 15 minutes, as mentioned earlier, we also tried setting the timeout to 30 minutes and did not observe any increase in the number of bugs demonstrating that increasing the time-out has diminishing returns. We also found the number of undetected bugs increase in the large contracts, as Mythril enumerates symbolic traces and this does not scale well in large contracts.

```
1 if (!addr.send (10 ether) || 1==1)
2   {revert();}
```

Figure 15: Unhandled exception code snippet 2.

Like the other tools, Mythril also misreported the types of many of the injected bugs. Figure 16 shows part of a buggy contract injected using *SolidiFI*. The injected contract allows users to manage their tokens and send tokens to each other. We injected a re-entrancy bug using *SolidiFI* in the contract at lines 185–188. However, Mythril reported the re-entrancy bug as “Unchecked Call Return Value” (i.e., Unhandled exception) at line 186. By inspecting this line of code, we can see that the return value of the `send` function is checked and the balance is reset to zero on the success of `send`, so there is no unhandled exception as reported. This calls into question Mythril’s soundness in detecting this type of bugs as well as its completeness in detecting Re-entrancy bugs.

```
177 function transfer(address _to, uint256 _value) public
178   returns (bool success) {
179     require(balances[msg.sender] >= _value);
180     balances[msg.sender] -= _value;
181     balances[_to] += _value;
182     emit Transfer(msg.sender, _to, _value);
183   }
184
185 function withdraw_balances_re_ent36 () public {
186   if (msg.sender.send(balances[msg.sender]))
187     balances[msg.sender] = 0;
188 }
```

Figure 16: Part of a buggy contract injected by reentrancy bug.

**Manticore** was not able to detect instances of re-entrancy and integer overflow/underflow. Unlike other evaluated tools employing symbolic executions, we noticed that Manticore takes a long time to analyze smart contracts, and in some cases it times-out. It consumes significant memory space as well on our system. Moreover, Manticore crashed and failed to analyze most of the contracts and threw exception errors. The 50 main contracts used in our experiments consist of 123 analyzable contracts (each contract file may contain more than one contract). Out of them, *Manticore* crashed for 83 contracts injected by re-entrancy bugs and 73 contracts injected by integer overflow bugs. We reached out to the tool developers

to get fixes or explanation for these issues; however, there was no response (as of the time of submission). For those challenges, we evaluated Manticore on detecting only two bug types.

**Securify** was not able to detect several cases of injected bugs belonging to re-entrancy, unchecked send, unhandled exceptions, and TOD. In addition, we found many cases where *Securify* failed to analyze the injected contracts and threw an error. Out of the 200 contracts injected by the four bug types (50 contracts for each bug type), *Securify* failed to analyze 5 contracts injected by unhandled exceptions, 4 injected by re-entrancy, 4 injected by unchecked send, and 4 injected by TOD bugs. If we excluded the injected bugs in those contracts, the number of undetected bugs by *Securify* will be as following: (re-entrancy: 105, unchecked send: 332, unhandled exceptions: 402, and TOD: 136). *Securify* also reported a high number of TOD false positives compared with *Oyente*. A recent study [27] found that the reported false alarms by *Securify* are due to over-approximation of the execution.

**Slither** Although *Slither* has almost 100% accuracy in detecting re-entrancy, timestamp, and tx.origin bugs, it was not able to detect many instances of unhandled exceptions. Moreover, it had high number of re-entrancy false positives as mentioned earlier.

## 8.2 Implications

**Tool Developers** There are two implications for tool developers. The first implication is that using pattern matching for detecting bugs, especially by employing simple approaches such as XPaths matching, is not an effective way for detecting smart contract bugs for the reasons mentioned earlier in this section. The second point is that bug detection approaches that are based on enumerating symbolic traces are impeded by path explosion and scalability issues. Therefore, there is a need for sophisticated analysis tools that also consider the semantics of the analyzed code instead of depending only on analyzing the syntax and symbolic traces. For example, static analysis might work better if combined with formal methods that consider the semantic specifications of Solidity and EVM. Recent papers [12, 14, 28–30] have proposed semi-automated formal verification for performing analysis of smart contracts.

**End Users of Tools:** For smart contract developers, who are the end users of the static analysis tools, there are three implications. First, they can use *SolidiFI* to assess the efficacy of static analysis tools to choose the most reliable tools with no or low false negatives for their use cases. Second, developers should not rely exclusively on static analysis tools, and should test the developed contracts extensively. *SolidiFI* can help them build test suites by introducing mutations and checking if the test cases can catch them. Finally, the generated bugs by *SolidiFI* and their relative locations in the code can be used for educating developers on writing secure code.

## 8.3 Limitations of *SolidiFI*

There are two limitations of *SolidiFI*. First, the current version of *SolidiFI* works only on Solidity static analysis tools. Although Solidity is the most common language for writing Ethereum smart contracts and most of the proposed tools target analysis of Solidity contracts, *SolidiFI* functionality can be easily extended to other languages. Secondly, the bug injection approach employed by *SolidiFI* requires pre-prepared code snippets (for each bug type), which requires

some manual effort. However, this is a one-time cost for each bug type (we have provided these as part of the tool).

## 8.4 Threats to Validity

An external threat to the validity is the limited number of smart contracts considered, namely 50. We have mitigated this threat by considering a wide-range of smart contracts with varying functionality and code sizes. We emphasize that *SolidiFI* covers all syntactic elements of Solidity up to version 0.5.12 and, our data-set contains a wide variety of contracts with different features (e.g., loops). Also, we selected contracts with different sizes that were representative of EtherScan contracts ranging from 39 to 741 locs, with an average of 242 loc (Table 2).

There are two internal threats to validity. First, the number of tools considered is limited to 6. However, as mentioned, these represent the common tools used in other studies on smart contract static analysis. Further, they are widely used in both academia and industry. All the tools available are open-source and are being actively maintained (with the exception of *Oyente*). Further, the implemented prototype of *SolidiFI* is reusable and can be easily extended to evaluate other tools. The second internal threat to validity is that we only injected 7 bug types. However, these bug types have been (1) considered by most of the tools evaluated, and (2) exploited in the past by real attacks. Therefore, we believe they are representative of security bugs in smart contracts.

Finally, a construct threat to validity is our measurement of false-negatives and false-positives. For false-negatives, it is possible that the bugs cannot be exploited in practice. We have partially mitigated this threat by sampling the set of false-negatives and attempting to exploit them (RQ3). For false-positives, it is possible that the reported bugs are true positives. Again, we have partially mitigated this threat by conservatively considering the bugs reported by the majority of the tools for each bug category as true positives.

## 9 RELATED WORK

**Bug-Finding Tools Evaluation** The approach of injecting bugs for evaluating the effectiveness of bug finders has been applied in other contexts than smart contracts. Bonett et al. proposed  $\mu$ SE [15], a mutation-based framework for the evaluation of Android static analysis tools that works as follows. First, a fixed set of security operators are created describing the unwanted behavior that the tools being evaluated aim to detect (e.g., data leakage). Then,  $\mu$ SE inserts the security operators into mobile apps based on mutation schemes, that consider Android abstractions, tools reachability and security goals of the tools, thereby creating multiple mutants that represent unwanted behavior within the apps. The mutated apps are analyzed using the static tools to be evaluated. However, unlike our work, the undetected mutants are analyzed manually. Further, their framework focuses only on data leak detection tools, unlike our work, which is more general.

Pewny et al. [40] automatically find potential vulnerable locations in C code, and modify the source code to make it vulnerable. Program analysis techniques are used to find sinks in the programs matching specific bug patterns, and find connections to user-controlled sources through data-flow. The program is modified accordingly to make it exploitable. In contrast to our approach, the

vulnerable code locations to be injected are randomly chosen, and the implemented prototype targets the injection of spatial memory errors through the modification of security checks.

Dolan-Gavitt et al. [23] proposed LAVA for generating and injecting bugs into the source code of programs using dynamic taint analysis. A guard is inserted for every injected bug for triggering the vulnerability if a specific value occurs in the input. Specifically, LAVA identifies an execution trace location where an unmodified and dead input data byte (DUA) is available. Then code is added at this location to make the DUA byte available, and use it execute the vulnerability. Unlike our work, the injection is based on dynamic taint analysis, and the injected bugs are accompanied by triggering inputs. In contrast, our goal is to transform invulnerable code to systematically introduce vulnerabilities in it.

In recent work, Akca et al. [11] proposed a tool that the authors used to compare the effectiveness of their introduced smart contracts static analyzer with some other tools. by injecting a single bug into the contract code. Unlike our approach that injects exploitable bugs into all potential locations in the contract code, the tool uses Fault Seeder [38] to generate contract mutants by injecting only a single bug snippet (hard-coded in the source code) into a specific location in the smart contract. In addition, the authors conducted manual inspection of the tool reports to determine false negatives. Injecting a single bug does not provide a comprehensive coverage evaluation of static analysis tools. Also, it is not clear how to evaluate the efficacy of static analysis tools on detecting deep vulnerabilities and corner cases by injecting only a single bug. As presented before, each bug can be introduced in the code in several ways, in this case, injecting a single-bug will not test the efficacy of the static analysis tools to detect various variants of each bug.

Durieux et al. [24] compared a number of smart contract static analysis tools. Unlike our work, the evaluation is based on using 69 manually annotated smart contracts with 112 bugs in total. The vulnerable contracts are collected from online repositories that are not agnostic of the evaluated tools, hence, results might be biased (e.g., collecting 50% of the contracts from SWC Registry referenced by Mythril and maintained by the team behind it). In contrast, our goal is to perform systematic and comprehensive coverage evaluation of static analysis tools by transforming invulnerable code to systematically introduce vulnerabilities into all valid locations. We evaluated 6 tools on detecting about 9369 distinct bugs. To provide a fair evaluation of the tools, we evaluate each tool only on the bugs that it is designed to detect. Our proposed approach can be easily used to evaluate smart contract static analysis tools for detecting other bug types. Further, it enables end-users to choose any dataset of smart contracts for evaluating the tools.

**Smart Contract Testing and Exploitation:** There have been many recent papers on testing smart contracts, and automatically generating security exploits on them. Wu et al. [45] produce test cases by mutating specific patterns in smart contracts. Chan et al. [18] develop a fuzz testing service (Fuse) to support the fuzz testing of smart contracts. This is a work in progress report, and only presents the architecture of the fuzz service being developed. Wang et al. [44] target the generation of test suites for smart contracts. This work guides automatic generation of test cases for Ethereum smart contracts. Eth-mutants [5] is another mutation testing tool for smart contracts. However, it is limited to replacing  $<$  to  $\leq$ , and  $>$  to

$\geq$  (and vice versa). Unlike our focus on evaluating smart contracts' static analysis tools, these papers target the generation of test cases for the smart contracts.

Other papers focus on automatic exploitation of smart contracts to generate exploits or malicious inputs to exploit found code vulnerabilities [27, 31, 32]. teEther [32] generates exploits for vulnerable contracts using symbolic execution with the Z3 constraint solver [22] to solve path constraints for the critical paths in the control flow graph. Contractfuzzer [31] uses the Application Binary Interface (ABI) specifications of vulnerable smart contracts to generate exploits (fuzzing inputs) for two vulnerabilities. SMARTSCOPY [27] also synthesizes adversarial contracts for exploiting vulnerabilities in contracts based on ABI specifications of the contracts covering larger set of vulnerabilities than Contractfuzzer. Echidna [20] has been proposed for fuzzing smart contracts. It supports grammar-based fuzzing to generate transactions to test smart contracts. The goal of these techniques is testing the smart contracts themselves for security vulnerabilities rather than testing bug-finding tools.

## 10 CONCLUSION

This paper proposed *SolidiFI*, a technique for performing systematic evaluation of Ethereum smart contract's static analysis tools based on bug injection. *SolidiFI* analyzes the AST (Abstract Syntax Tree) of smart-contracts and injects pre-defined bug patterns at all possible locations in the AST. *SolidiFI* was used to evaluate 6 smart contract static analysis tools, and the evaluation results show several cases of bugs that were not detected by the evaluated tools even though those undetected bugs are within the detection scope of tools. *SolidiFI* thus identifies important gaps in current static analysis tools for smart contracts, and provides a reproducible set of tests for developers of future static analysis tools. It also allows smart contract developers to understand the limitations of existing static analysis tools with respect to detecting security bugs.

Future work will consist of expanding *SolidiFI* to other smart contract languages than Solidity, and automating the bug definition processes for injecting new bug types.

## ACKNOWLEDGMENTS

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and a research gift from Intel. We thank Julia Rubin, Sathish Gopalakrishnan, Konstantin Beznosov, and the anonymous reviewers of ISSTA'20 for their helpful comments about this work.

## REFERENCES

- [1] 2016. Analysis of the DAO exploit. <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit>
- [2] 2017. History of Ethereum Security Vulnerabilities, Hacks, and Their Fixes. <https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes>
- [3] 2017. The parity wallet breach. <https://bitcoincexchangeguide.com/parity-wallet-breach>
- [4] 2017. Remix - Solidity IDE. <http://remix.ethereum.org>
- [5] 2018. eth-mutants. <https://github.com/federicobond/eth-mutants>
- [6] 2018. New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). <https://medium.com/@peckshield/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536>
- [7] 2020. CVE-2018-10299 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2018-10299>
- [8] 2020. INFURA. <https://infura.io>
- [9] 2020. MetaMask. <https://metamask.io>

- [10] 2020. solidity-security-blog. <https://github.com/sigp/solidity-security-blog>
- [11] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 482–489.
- [12] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 66–77.
- [13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*. Springer, 164–186.
- [14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Golamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.
- [15] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2018. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1263–1280.
- [16] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv preprint arXiv:1809.03981* (2018).
- [17] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. URL <https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-White-Paper> 7 (2014).
- [18] WK Chan and Bo Jiang. 2018. Fuse: An Architecture for Smart Contract Fuzz Testing Service. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 707–708.
- [19] Christopher D Clack, Vikram A Bakshi, and Lee Braine. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771* (2016).
- [20] Crytic. [n.d.]. Echidna. <https://github.com/crytic/echidna>
- [21] Chris Dannen. 2017. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Springer.
- [22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340.
- [23] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 110–121.
- [24] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2019. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. *arXiv preprint arXiv:1910.10601* (2019).
- [25] Etherscan. [n.d.]. Etherscan. <https://etherscan.io>
- [26] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
- [27] Yu Feng, Emina Torlak, and Rastislav Bodík. 2019. Precise Attack Synthesis for Smart Contracts. *CorR abs/1902.06067* (2019). arXiv:1902.06067 <http://arxiv.org/abs/1902.06067>
- [28] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.
- [29] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *Kevm: A complete semantics of the ethereum virtual machine*. Technical Report.
- [30] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.
- [31] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.
- [32] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. {USENIX Association}, 1317–1333.
- [33] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.
- [34] Florian Mathieu and Ryno Matthee. 2017. Blockfix: decentralized event hosting and ticket distribution network. <https://www.cryptoground.com/storage/files/1527588859-blockfix-wp-draft.pdf>
- [35] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.
- [36] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HTB SECCONF Amsterdam* (2018).
- [37] Reza M Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 103–113.
- [38] Chao Peng, Sefa Akca, and Ajitha Rajan. 2019. SIF: A Framework for Solidity Contract Instrumentation and Analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 466–473.
- [39] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? *arXiv preprint arXiv:1902.06710* (2019).
- [40] Jannik Perny and Thorsten Holz. 2016. EvilCoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 214–225.
- [41] Ferdinand Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, Premkumar T Devambu, et al. 2012. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 50–59.
- [42] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhayev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. (2018).
- [43] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buerzli, and Martin Vechev. 2018. Security: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.
- [44] Xingya Wang, Haoran Wu, Weisong Sun, and Yuan Zhao. 2019. Towards Generating Cost-Effective Test-Suite for Ethereum Smart Contract. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 549–553.
- [45] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. 2019. Mutation testing for ethereum smart contract. *arXiv preprint arXiv:1908.03707* (2019).

# A Programming Model for Semi-implicit Parallelization of Static Analyses

Dominik Helm

Florian Kübler

Jan Thomas Kölzer

helm@cs.tu-darmstadt.de

kuebler@cs.tu-darmstadt.de

jan.koelzer@stud.tu-darmstadt.de

Technische Universität Darmstadt

Department of Computer Science

Germany

Philipp Haller

phaller@kth.se

KTH Royal Institute of Technology

School of Electrical Engineering and

Computer Science

Sweden

Michael Eichberg

Guido Salvaneschi

Mira Mezini

mail@michael-eichberg.de

salvaneschi@cs.tu-darmstadt.de

mezini@cs.tu-darmstadt.de

Technische Universität Darmstadt

Department of Computer Science

Germany

## ABSTRACT

Parallelization of static analyses is necessary to scale to real-world programs, but it is a complex and difficult task and, therefore, often only done manually for selected high-profile analyses. In this paper, we propose a programming model for semi-implicit parallelization of static analyses which is inspired by reactive programming. Reusing the domain-expert knowledge on how to parallelize analyses encoded in the programming framework, developers do not need to think about parallelization and concurrency issues on their own. The programming model supports stateful computations, only requires monotonic computations over lattices, and is independent of specific analyses. Our evaluation shows the applicability of the programming model to different analyses and the importance of user-selected scheduling strategies. We implemented an IFDS solver that was able to outperform a state-of-the-art, specialized parallel IFDS solver both in absolute performance and scalability.

## CCS CONCEPTS

- Theory of computation → Program analysis; Parallel computing models;
- Software and its engineering → Automated static analysis.

## KEYWORDS

static analysis, concurrency, parallelization

### ACM Reference Format:

Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. 2020. A Programming Model for Semi-implicit Parallelization of Static Analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397367>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397367>

## 1 INTRODUCTION

Parallelization of static analyses is a promising avenue (a) for making analyses scale to ever-growing code bases, and (b) for shortening development cycles by reducing delays caused by build pipelines and continuous integration. By leveraging modern multi-core CPUs and GPUs, parallelization can also increase energy efficiency, e.g., when executing static analyses on virtualized cloud infrastructures. However, parallelizing static analyses is challenging, especially since advanced static analyses, e.g., general points-to [4], advanced call graph [8], or purity [14] analyses, are not data-parallel.

Relying on shared-memory threads and conventional synchronization primitives, such as locks and monitors, makes parallelization of static analyses cumbersome and requires significant expertise to ensure correctness, while, nevertheless, often not fully exploiting all available hardware parallelism [10].

Existing approaches to parallel static analysis have addressed the problem in two limited ways. Some have done so by targeting only specific high-profile analysis techniques [25, 26, 29], respectively specific frameworks [33]. Others require fundamental rethinking of the implementation strategy (e.g., avoiding explicit worklists) in order to adjust to the parallelism model of GPUs [25, 29]. For example, the approach of [25] requires (a) an adaptation of the data structures used by the analysis to the GPU memory model, (b) an explicit distribution of work to threads, and (c) novel concurrent algorithms amenable to the execution on GPUs. Enabling parallelization of static analyses *in a generic way*, i.e., not specialized to a single analysis or framework, and *implicit* [13], i.e., without requiring to restructure analysis algorithms or even devise completely new ones, remains an open challenge.

In this paper, we propose a new approach for deterministic parallel execution of static analyses. The approach offers a programming model not bound to any specific analysis kind (*analysis-independent*) and providing *semi-implicit parallelization* [23], i.e., not requiring invasive changes to common implementation strategies for static analyses in order to parallelize their execution. The exposure of the analysis writer to aspects relevant to parallelism is minimal; specifically, there is no need to (a) modify data structures used by the analysis or (b) to distribute work explicitly to threads.

The analysis-independent, semi-implicit parallelization is enabled by a declarative reactive programming model. Such a programming model provides high-level constructs for declaring dependencies between analysis results, while automating the propagation of updated analysis results across dependency chains.

All propagations are concurrent, except for those marked explicitly as sequential by the user if they share mutable state, thus avoiding non-determinism due to race conditions. In addition, the domain of analysis results is restricted to be a (semi-)lattice to ensure *determinism of concurrent updates*. This is, however, not really a restriction, since even complex, state-of-the-art static analyses compute results representable in bounded (semi-)lattices. Declaration of sequential updates and the requirement to use (semi-)lattices are the only aspects of parallelism exposed to analysis writers.

Declarative dependencies enable defining propagation strategies that are tuned to the domain of static analyses. For example, certain analyses benefit from a prioritization of updates where the more (source) dependencies a computation has, the lower the priority of that computation is; such a prioritization allows “batching” updates such that a larger number of more-up-to-date values is queried before a target computation is performed. Our approach also enables analysis writers to plug in analysis-specific strategies.

The contribution of this paper is to provide language abstractions that enable developers to formulate static analyses as a network of cells with dependencies among them. This approach allows the framework to handle concurrency semi-automatically while at the same time being applicable to a wide range of static analyses. Building on our previous work [10], we introduce four important concepts: (a) sequential cells that enable analyses to pass mutable state between callback invocations; (b) custom cell updaters that enable omitting costly join operations; (c) pluggable scheduling strategies that can be adapted to the needs of each analysis; and (d) aggregation of updates in order to reduce computation overhead.

The approach is implemented as a concurrency library in Scala, called Reactive Async 2 (RA2). We open-sourced the implementation on GitHub.<sup>1</sup>

We evaluate RA2 along two dimensions. First, we give evidence on the approach being analysis-independent and enabling semi-implicit parallelism by applying it to two different kinds of analyses (see Section 3.1), a purity analysis and the IFDS framework [32].

Second, we empirically evaluate the performance and scalability of RA2 (Sections 3.2 to 3.4). Our evaluation shows that the performance scales well w.r.t. the number of processor cores. With 10 threads on a 10-core CPU, we achieve speed-ups greater than 4.8x and, hence, successfully parallelize over 88% of the computation according to Amdahl’s law. A comparison of different scheduling strategies shows performance differences up to 1100%, suggesting that the choice of scheduling strategy is of great importance. On a single core, our parallel IFDS is only 28% slower than an optimized sequential implementation using the fixed-point computation framework provided by the static analysis framework OPAL [6, 28]. Already with 2 threads the parallel implementation runs 1.3x faster than OPAL. Compared to the state-of-the-art IFDS solver Heros [3, 15], RA2 has comparable single-thread performance (RA2 is 15% faster) and consistently achieves significantly higher

speed-ups. While Heros achieves a maximum speed-up of 2.36 using 8 threads, the speed-ups of RA2 are 3.53 using 8 threads and continues to improve for higher thread counts. A thorough comparison to an actor-based parallel IFDS solver [33] was unfortunately not possible as the implementation of that system is not available, but the speed-ups are comparable.

The rest of the paper is organized as follows: Section 2 presents the programming model including an in-depth example of a simple analysis. The approach is evaluated in Section 3. Section 4 presents potential threats to validity to our evaluation. We conclude the paper discussing related work in Section 5 and providing a summary and outlook in Section 6.

## 2 APPROACH

In this section, we present our approach to analysis-independent and semi-implicit parallelization of static analyses. We start with a high-level overview of the programming model that the approach imposes (2.1) followed by advanced concepts to ensure correctness in the presence of concurrent updates and shared mutable state (2.2). Next, we introduce RA2’s solver, which performs the parallelization and resolves (cyclic) dependencies (2.3). We present pluggable scheduling strategies, which enable analysis-specific tuning and aggregation of updates with the goal to improve on performance (2.4) and finally give an in-depth example of a simple purity analysis implemented in RA2 (2.5).

### 2.1 Programming Model Basics

To illustrate how analyses are implemented in RA2, assume we want to develop a method purity analysis determining whether a method is deterministic and free of side-effects (see, e.g., [14]).

The goal of any static analysis is to compute a specific property for a given entity. In the example of the purity analysis, methods are the entities and the property is the purity, which can have one of two values: pure or impure. In RA2, each property must form a lattice (or at least a partial order with a bottom element), common data structures for properties calculated by static analyses. The property of an entity may depend on some other properties of related entities. In a purity analysis, if `foo` calls `bar`, the computation of `foo`’s purity requires the purity of `bar`. We say `foo` depends on/is a *depender* of `bar`, or `bar` is a *dependee* of `foo`.

In RA2, analyses are implemented as two functions—an *initial analysis function* and a *continuation function*. Both are invoked by the reactive framework underlying RA2.

Given an entity, the initial analysis function computes the initial property of that entity based on the information already available, i.e., local information, such as the source code, and the current property values of dependees. The initial analysis function also queries and collects the dependencies of an entity. The dependencies may have a non-final property value (or none at all) at the time the initial function is executed. The list of dependencies is used to get notified of potential future changes of their property values. These changes may in turn lead to updates of the entity’s property value. The initial analysis function returns the initial property and registers the dependencies along with a continuation function.

A continuation function of an entity `e` is invoked by RA2 whenever dependencies of `e` (e.g., the purity information for a called

<sup>1</sup><https://github.com/phaller/reactive-async>

method) change. Its result defines how the property value of  $e$  is to be updated, whereby the updated value must be a monotonic refinement according to the property's lattice. Unless the continuation function declares its result as final, it will be invoked again for further updates of dependees. Both the initial analysis and invocations of the continuation functions are *tasks* that RA2 executes concurrently.

Analysis results and dependencies are maintained in *cells*. When implementing a static analysis, we create one cell per pair of analyzed entity and property. Cells are shared by the different concurrent tasks. Every cell is explicitly associated with the *lattice* of the property values that it manages (e.g., the cells in Listing 1 are associated with the *Purity* lattice). A cell that was created to store purity information therefore cannot be used—at some later point in time—to store data-flow information.

Dependencies are created by connecting two cells using the continuation function—to respond to updated cells, continuation functions are used as callbacks. To ensure that a cell is notified about the update of its dependees, its dependencies are explicitly declared and registered using the *when* method. For instance, `cell2.when(cell1)(continuation)` registers `cell1` as a dependee of `cell2`, the function *continuation* is called *when* the value of `cell1` changes.

The continuation function processes the changed dependee's (`cell1` in our case) new value and returns an *Outcome* object, which decides whether and how the dependent cell (`cell2` in our case) should be updated. RA2 provides three types of *Outcome* objects: A `NextOutcome(v)` result means that the dependent cell should be updated with value  $v$  according to the specified updater (cf. Section 2.2). `FinalOutcome(v)` states that additionally this update is final. If `NoOutcome` is returned, the value of `cell2` is not changed at all.

To illustrate the use of cells and continuations, consider Figure 1 which graphically depicts the cells and dependency from Listing 1 and the propagation of an update for this dependency. Assume that the two cells, `cell1` and `cell2`, have been initialized with `Pure` (A) through the use of `NextOutcome`, i.e., their values can still change. In Lines 4 to 7, a dependency between `cell1` and `cell2` is introduced (B). Recall that for a purity analysis, this is necessary if the method represented by `cell2` invokes the one represented by `cell1`. Whenever `cell1` is updated (using `FinalOutcome` or `NextOutcome`), the continuation is executed and the returned *Outcome* is used to update `cell2`. Consider the case when a final update for `cell1`—with the value `Impure`—occurs (C). This will cause the continuation to be invoked with `Impure` as an argument (the new value of `cell1`) (D). Since a method that calls an impure method is also impure, the continuation returns a `FinalOutcome` with value `Impure`. Thus, `cell2` is completed with value `Impure` (E), and the dependency is removed (F). If the continuation returned a `NextOutcome`, the update of `cell2` would be an intermediate one.

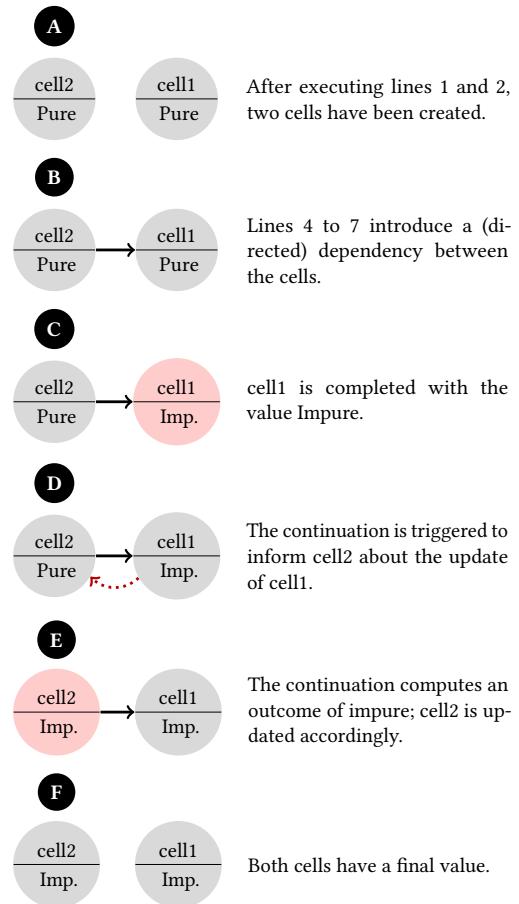
## 2.2 Advanced Constructs for Correctness

To ensure correctness and termination, cell updates must have a well-defined semantics. RA2 executes updates concurrently, and thus without a guaranteed order. Yet, monotonic updates ensure determinism regardless of the order. Mutable stated shared between the continuations of a cell may also lead to non-determinism when the updates are executed concurrently.

**Listing 1: Example of dependencies and continuations**

```

1 val cell1: Cell[Purity] = ...
2 val cell2: Cell[Purity] = ...
3
4 cell2.when(cell1) { update =>
5   if (update.head.get.value == Impure) FinalOutcome(Impure)
6   else NoOutcome
7 }
```



**Figure 1: Execution of Listing 1**

*Monotonic Updates.* To guarantee determinism of cell updates, they must be monotonically increasing according to the underlying lattice. RA2 therefore encapsulates cell updates in so-called *updater* objects that determine how updates for the cell are processed.

Monotonicity can be automatically guaranteed by using the *join* operator of a cell's lattice to aggregate new values with the previous value of the cell. The outcome returned by each continuation is joined with the cell's previous value to compute the least upper bound, which then becomes the cell's new value. RA2 provides the updater type *AggregationUpdater* that offers this semantics. However, there are several good reasons for supporting other updater semantics. In general, performing joins can be expensive when dealing with lattices that are not based on singleton values, e.g.,

sets. Complex analyses may already have to perform the *join* explicitly as part of the continuation function, thereby guaranteeing the monotonicity by design and making another implicit join during the cell update obsolete. To cover such needs, RA2 provides the `MonotonicUpdater` as a drop-in replacement for the `AggregationUpdater`.

The `MonotonicUpdater` does not perform a *join* operation, but only checks whether the given update fulfills monotonicity. This check is defined by the lattice. For expensive checks, it can be used only during development and simplified or disabled in production when the analysis is known to guarantee monotonicity. The `MonotonicUpdater` also allows for using partial orders instead of lattices. The partial orders, however, still require a *bottom element* and must fulfill the ascending chain condition, i.e., monotonically increasing operations must converge eventually – we use this kind of updater, e.g., for our IFDS solver (see Section 3). As the IFDS solver’s computations are required to be performed sequentially and only introduce additional flow edges, updates are guaranteed to be monotonic and no other (potentially expensive) join operation is required.

Which updater should be used for a specific cell can be defined when creating the cell by the `HandlerPool` – the interface of the analysis implementations to the underlying reactive system of RA2, which is presented in Section 2.3.

*Sequential Updates.* Shared mutable state affected by updates needs to be thread-safe. Advanced analyses require maintaining mutable state between cell updates. For example, an IFDS analysis keeps track of already computed path edges in order to extend them once updates on the analyzed method’s callees become available. Mutable state can also be used to explicitly keep track of the set of dependencies. Updates that affect such mutable state need to be sequential. Otherwise, continuations for several incoming updates could be executed concurrently, leading to non-determinism in the presence of mutable state due to race conditions.

To enable the thread-safe use of mutable state, RA2 provides cells with *sequential updates*, whose continuations are ensured to run sequentially. As such, cells with sequential updates free developers from the need to use locks or other concurrency mechanisms to protect shared mutable state. The example in Listing 2 illustrates cells with sequential updates. Those cells are created using the `mkSequentialCell` method of the `HandlerPool` (cf. Section 2.3). While `dependee1` and `dependee2` may be updated concurrently, RA2 ensures that all callbacks targeting `seqCell11`—`continuation1` and `continuation2`—are invoked sequentially (though with no guarantee on the relative order). Hence, both callbacks may safely access shared state – as long as that state is only shared among callbacks targeting `seqCell11`. To still exploit the benefits of parallel computations, `continuation3` is run concurrently, even with `continuation1` being triggered by the same dependee, `dependee1`. Hence, `continuation1` and `continuation2` must not share state with `continuation3`, as the latter targets a different cell.

### 2.3 Handler Pool

A central unit of RA2 is the `HandlerPool`. It creates cells and manages the execution of initial functions and the propagation of updates along cell dependency chains by executing respective continuations. It implements the parallelization which analyses can use out-of-the-box and do not need to implement on their own. For each initial

**Listing 2: Example of sequential updates**

```

1  val seqCell11 = pool.mkSequentialCell(...)
2  val seqCell12 = pool.mkSequentialCell(...)
3
4  seqCell11.when(dependee1)(continuation1)
5  seqCell11.when(dependee2)(continuation2)
6  seqCell12.when(dependee1)(continuation3)
```

analysis function and each triggered continuation, the runtime system creates a task that is eventually executed by an idle thread. The `HandlerPool` keeps track of all active threads and all tasks being registered for execution and schedules their execution.

The `HandlerPool` is RA2’s fixed-point solver, as such it also resolves situations where the execution gets stuck. As it keeps track of running and pending tasks, the pool is able to detect *quiescence*. The system is quiescent when there are no unfinished submitted tasks currently queued or running. However, even when quiescence is reached, not all cells are guaranteed to contain final results. This is true in the following cases:

- (1) A chain of dependencies such that each cell’s result depends on another cell’s result where these dependencies form a cycle is called *cyclic dependency*. A simple example are two cells *A* and *B*, where *A* depends on *B* and *B* depends on *A*. Such a cyclic dependency where each cell in the cycle solely depends on cells of that cycle is called a closed strongly connected component (cSCC).
- (2) Cells that do not depend on any other cells are called *independent*. Independent cells that have not been completed are referred to as *independent unresolvable cells* (IUC). An IUC arises when all dependees of a cell have been completed (and the corresponding dependencies have therefore been dropped), but the cell itself was not completed yet. This happens if on the last invocation of the continuation, that continuation—without tracking dependencies explicitly—cannot recognize that there will not be any future invocations. In the example of Figure 1, `ce112` becomes independent after the final update of `ce111` as the dependency between `ce112` and `ce111` is removed by the system (F). This shows the case, when `ce112` is completed. But, if `ce111` had instead been completed with a result property value *pure*, the continuation would have returned `NoOutcome` and, therefore, `ce112` would not be completed. As the dependency would still be removed, `ce112` would become an IUC.

Once cycles and independent cells are detected, two methods, that are implemented by the analysis designer when specifying the lattice, are used to resolve them: `resolve` and `fallback`. The `resolve` method takes a list of all cells in one cSCC and returns for each cell the final value it should be resolved to. Resolving one cSCC does not trigger callbacks of cells in that cSCC, but does so for their dependents outside of it. The `fallback` method works similarly, but is given a list of IUCs. Different to the cells of a cSCC, the IUCs must be resolved independently of each other. Each cell is then completed with the returned associated value, which is typically the cell’s current value.

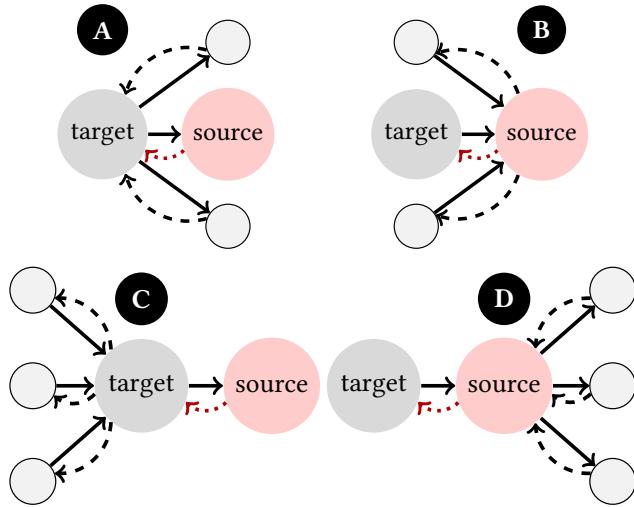


Figure 2: Scheduling Strategies

## 2.4 Scheduling

The order in which individual tasks are scheduled can be of utmost importance in order to provide effective parallelization for a specific (analysis) domain [33]. In the case of IFDS-A, scheduling strategies that prioritize values that might have a bigger impact were shown to be beneficial. Thus, different scheduling strategies are needed.

The HandlerPool is parametric in the scheduling strategy to enable an analysis designer to plug in a scheduling strategy to influence the order in which tasks ready for execution are picked up. Whenever a task is submitted to the pool for execution, the scheduling strategy is invoked to calculate a priority for the dependency in question. This priority is used in a priority queue that tracks all tasks that may be executed concurrently.

Dependency continuations that target sequential cells must not run concurrently. To ensure this, each sequential cell keeps track of all tasks updating it. Again, this is done via a priority queue that uses the developer-supplied scheduling strategy. Tasks can be dequeued from this queue in the order of respective priority, hence respecting the scheduling strategy.

Besides a default last-in-first-out scheduling strategy with first-in-first-out work stealing, RA2 provides several other general-purpose scheduling strategies out-of-the-box. These are applicable to any kind of analysis as they only take into account how many dependees a cell has and how many cells (directly) depend on the value that is being updated via the continuation. These strategies are illustrated in Figure 2 (A–D). As in Figure 1, straight arrows are dependencies, while discontinuous arrows represent potential update messages. The strategies determine the priority of the message from source to target (dotted red message in Figure 2).

- (A) **TargetsWithManySourcesFirst/Last.** The higher the number of cells (in addition to source) that target depends on, the higher/lower the priority.
- (B) **SourcesWithManyTargetsFirst/Last.** The higher the number of cells depending on source (in addition to target), the higher/lower the priority.

### Listing 3: A scheduling strategy for lattice values

```

1  case class LatticeValueStrategy[V](prioritizedValue: V)
2  extends SchedulingStrategy[V] {
3    override def calcPriority(
4      tgt: Cell[V], src: Cell[V], v: Outcome[V]
5    ): Int = calcPriority(tgt, v)
6
7    override def calcPriority(
8      tgt: Cell[V], v: Outcome[V]
9    ): Int = v match {
10      case FinalOutcome(prioritizedValue) => -1
11      case _ => 1
12    }
13 }
```

- (C) **TargetsWithManyTargetsFirst/Last.** The higher the number of cells depending on target, the higher/lower the priority.
- (D) **SourcesWithManySourcesFirst/Last.** The higher the number of cells that source depends on, the higher/lower the priority.

The generic-purpose strategies are simple and only take the (local) shape of the dependency graph into account. Nonetheless, they can influence the analysis' behavior significantly. Some strategies (the *xFirst* ones) try to prioritize updates that may have a greater impact on the result by influencing many cells. This may lead to faster stabilization of the result. Other strategies (the *xLast* ones) delay such influential updates, providing more opportunities to aggregate them (cf. below), potentially reducing the overall number of updates required. Our evaluation (cf. Section 3.2) shows that this has a significant impact on scalability and performance.

Scheduling strategies also allow to encode and take into consideration domain-specific knowledge about the relevance of different property values. For our purity analysis, propagating `Impure` may be more relevant to target cells than propagating `Pure`. This is because a call to an impure method is impure, thus a single impure dependee will result in the cell being completed. In contrast, a call to a pure method does not change the outcome as long as there are other dependees that might be impure. To accelerate the propagation of specific lattice values, we can use a strategy that returns a higher priority for the respective continuations.

The implementation of such a strategy prioritizing a given lattice element is shown in Listing 3. RA2 provides two functions to calculate the priority for scheduling: the first one is used in scheduling continuations while the second one is used for resolving cycles and IUCs where no source cell exists. Note that, based on Java's priority queues, a low value will prioritize the task to be scheduled early.

As the invocation of continuations may be delayed according to prioritization, the cell that triggered a continuation may be updated again before the continuation is actually invoked. This enables an important optimization: instead of invoking the continuation with the first updated value and later invoking the continuation again, the continuation will be invoked only once using the most recent value. RA2 can also aggregate updates from multiple sources that are passed to the continuation as a list of updates. Both kinds of aggregations can reduce the overhead of continuation invocations. This also has a transitive effect: less invoked continuations produce less intermediate results that in turn trigger less continuations.

## 2.5 Reactive Async 2 at Work

In the following, we demonstrate how to apply the proposed programming model to implement a very simple purity analysis. We show the complete analysis, including the lattice definition, the initial analysis function, its continuation, and how to bootstrap the analysis, leaving out just minor details, e.g., error handling.

**Listing 4: A simple lattice for purity information**

```

1 sealed trait Purity
2 case object Pure extends Purity
3 case object Impure extends Purity
4
5 object Purity {
6   implicit object PurityLattice extends Lattice[Purity] {
7     override def join(v1: Purity, v2: Purity): Purity = {
8       if (v1 == Impure) Impure else v2
9     }
10
11   override val bottom: Purity = Pure
12 }
13 }
```

Listing 4 shows the specification of the lattice, including the bottom value and the *join* function. The lattice has two elements, namely *Pure* (its bottom element) and *Impure*.

The analysis function shown in Listing 5 computes the purity of a given method by checking its instructions. If there is an instruction affecting the purity, e.g., a static field write, the method is immediately considered impure. Additionally, we consider native methods, methods with reference type parameters, or non-monomorphic (i.e., virtual and interface) method calls as impure. For method calls that are not self-recursive, the callee's cell is added as a dependency. If such callee would be impure, the analyzed method itself would be impure. With a call graph available, polymorphic calls could simply be handled by adding a dependency for each possible callee. After checking all instructions, the method is found to be pure, and can only be refined to impure by an update of a dependency. To cover the latter case, we use *when* to register the continuation function to react on updates for these dependencies.

The continuation function handling updates of dependencies is shown in Listing 6. As stated above, impure callees lead to an impure method. We take advantage of RA2's aggregation of updates: The continuation function may receive updates for multiple dependents at once and if a single one is impure, the cell is completed.

Listing 7 shows the fallback and cycle-resolution strategies as explained in subsection 2.4. For the purity analysis, all impure values will be marked as final immediately. Therefore, unresolved cells must be *Pure* for both, cycles and IUCs.

Listing 8 shows how to (1) initialize and (2) start the analysis, (3) await its completion, and (4) retrieve the results. Parallelization is done by the *HandlerPool* – the number of threads is set explicitly in this example. We also specify the scheduling strategy here, using the lattice-based strategy from Listing 3 to prioritize updates of of impure methods. Cells are created through the *HandlerPool* and their initial analysis is then triggered. In particular, note how we specify the *AggregationUpdated* for each cell (line 11). This explicit specification is for demonstration only, as aggregating updates is the default in RA2. Triggering the cells starts the concurrent

**Listing 5: Determine the purity of a method**

```

1 def analyze(method: Method): Outcome[Purity] = {
2   val cell = methodToCell(method)
3
4   if (method.isNative || method.hasReferenceTypeParameter())
5     return FinalOutcome(Impure)
6
7   val dependencies = mutable.Set.empty[Cell[Purity]]
8   for (instruction <- method.instructions) {
9     instruction match {
10       case gs: GETSTATIC =>
11         resolveFieldReference(gs) match {
12           case Some(field) if field.isFinal =>
13             /* Constants do not impede purity */
14           case _ =>
15             return FinalOutcome(Impure)
16         }
17
18       /* For simplicity: only handle monomorphic calls */
19       case INVOKESTATIC | INVOKESTATIC =>
20         resolveNonVirtualCall(instruction) match {
21           case Success(callee) =>
22             /* Self-recursive calls do not impede purity */
23             if (callee != method)
24               dependencies.add(methodToCell(callete))
25
26           case _ /* Unknown callee */ =>
27             return FinalOutcome(Impure)
28         }
29
30       case NEW | PUTSTATIC | ... =>
31         return FinalOutcome(Impure)
32
33       case _ =>
34         /* All other instructions are pure. */
35     }
36   }
37
38   if (dependencies.isEmpty) {
39     FinalOutcome(Pure)
40   } else {
41     cell.when(dependencies)(continuation)
42     NextOutcome(Pure)
43   }
44 }
```

**Listing 6: Continue with updates for callees**

```

1 def continuation(
2   v: Iterable[(Cell[Purity], Outcome[Purity])])
3 ): Outcome[Purity] = {
4   if (v.exists(_._2 == FinalOutcome(Impure)))
5     FinalOutcome(Impure)
6   else
7     NoOutcome
8 }
```

computation of the initial analysis functions. After quiescence has been reached, the cells contain the final results.

## 3 EVALUATION

Our evaluation aims to answer the following research questions:

- RQ1: Is RA2 applicable across different kinds of static analyses to enable semi-implicit parallelization?

**Listing 7: Resolving cycles and IUCs**

```

1 object PurityKey extends Key[Purity] {
2     def resolve(
3         cells: Iterable[Cell[Purity]]
4     ): Iterable[(Cell[Purity], Purity)] = {
5         cells.map(cell => (cell, Pure))
6     }
7
8     def fallback(
9         cells: Iterable[Cell[Purity]]
10    ): Iterable[(Cell[Purity], Purity)] = {
11        cells.map(cell => (cell, Pure))
12    }
13 }

```

**Listing 8: Setting up and starting the analysis.**

```

1 def main(project: Project): Unit = {
2     // 1. Initialize HandlerPool and Cells
3     val pool: HandlerPool[Purity] = new HandlerPool(
4         key = PurityKey,
5         parallelism = 10,
6         schedulingStrategy = LatticeValueStrategy(Impure)
7     )
8     var methodToCell = Map.empty[Method, Cell[Purity]]
9     for (method <- project.allMethods) {
10         val cellCreator = pool.mkCell(_ => analyze(method))
11         val cell = cellCreator(AggregationUpdater)
12         methodToCell += method -> cell
13     }
14
15     // 2. Start analyses
16     for (method <- project.allMethods) {
17         methodToCell(method).trigger()
18     }
19
20     // 3. Wait for completion
21     val fut = pool.quiescentResolveCell()
22     Await.ready(fut, 30.minutes)
23     pool.shutdown()
24
25     // 4. Retrieve results
26     val pureMethods =
27         methodToCell.filter(_.getResult() == Pure).keys
28 }

```

- RQ2: How do scheduling strategies affect performance?
- RQ3: What is the overhead compared to sequential analyses?
- RQ4: How does the RA2-based IFDS implementation compare to other state-of-the-art IFDS analyses?

For the evaluation, we used a machine with an Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz (10 cores / 20 threads) and 128 GB RAM. The OS running is Ubuntu 18.04.3. The analyses were run using OpenJDK 1.8\_212. The JVM was started with 16 GB of heap memory (-Xmx16G). We analyzed the JRE 1.7.0 update 95 from the publicly available Doop benchmarks project [5] to ensure repeatability. The purity analysis was executed on the complete JRE, while the IFDS analysis was executed on the runtime jar only. For each experiment we report the median runtime of seven executions.

While RA2 itself is framework independent, we used the OPAL framework [6] to provide bytecode parsing and an intermediate representation for our analyses to work upon.

**3.1 RQ1: Applicability**

To answer RQ1, we take another look at our purity case study from Section 2.5. As it shows, the implementation of a simple purity analysis, including definition of the lattice and an execution harness takes just about 80 lines of code (excluding whitespace and comments). We observe that RA2’s programming model allows implicit parallelization in this case: No explicit handling of parallelization, e.g., creation of threads, locks, or tasks, is required by the actual analysis<sup>2</sup>. In particular, no rethinking of the algorithm is required, as it is, e.g., for GPU-based solutions [25, 29] to map the execution to the parallel hardware. Extending the analysis to a more powerful purity analysis would require changes only to the analysis and continuation functions (and potentially an extension of the lattice for a finer granularity) and would not introduce any additional complexity related to the parallelization.

For a more complex analysis, we implemented a solver<sup>3</sup> for IFDS. This implementation was adapted from the IFDS solver in the OPAL framework with only minor changes to support the different dependency handling. IFDS (cf. Appendix) is a general framework for dataflow analyses and has been implemented numerous times (e.g., in WALA [39], Heros [3], Flix [21] or IFDS-A [33]). It has been parallelized in the past and we evaluate our performance against the state-of-the-art IFDS solver Heros. IFDS-A’s implementation was never publicly available, however. WALA and Flix do not even have a parallel implementation. Flix’s implementation in particular is only a proof-of-concept that ceased to work with current versions of Flix.

Our implementation makes use of MonotonicUpdaters, which reduces the number of large set operations. It requires sequential updates as it maintains mutable state between the continuation invocations to keep track of dataflow edges already known. Thus, the parallelization is semi-implicit in the case of the IFDS solver.

Using our IFDS solver, we implemented a taint analysis as a client analysis used in the rest of the evaluation. This analysis is inspired by FlowTwist [20] and identifies public or protected methods in the Java Runtime 7 (rt.jar) with return type java.lang.Object or java.lang.Class that have a string parameter that is later used in an invocation of java.lang.Class.forName. If such flows are found, attackers may get the ability to load a class of their choice. The analysis tracks local variables and is field-sensitive.

RA2 is able to provide (semi-)implicit parallelization to the simple purity analysis as well as the significantly more complex IFDS solver. Both analyses are very different in their kind and there is no specialized support for any of them implemented in RA2, indicating that RA2 is applicable to different kinds of static analyses.

**3.2 RQ2: Scheduling Strategies**

To answer RQ2, we evaluated different scheduling strategies for both the IFDS and purity analysis introduced above. The execution times reported are for ten threads each, corresponding to the number of physical cores of our system.

*IFDS Analysis.* We evaluated the performance for the analysis-independent strategies. Table 1 shows the median execution times

<sup>2</sup>Only within the main function one Future needs to be awaited.

<sup>3</sup><https://github.com/phaller/reactive-async/tree/v2.0.0/core/src/test/scala/com/phaller/rasync/test/opal/ifds>

**Table 1: Performance of scheduling strategies for IFDS**

Speed-up shown compared to (a) default and (b) slowest strategy.

Strategy	Run time [s]	Speed-up (a)	Speed-up (b)
DefaultScheduling	27.29	0.00%	13.3%
SourcesWithManyTargetsLast	21.00	23.1%	33.3%
TargetsWithManySourcesLast	19.84	27.3%	37.0%
TargetsWithManyTargetsLast	29.31	-7.40%	6.86%
SourcesWithManySourcesLast	21.40	21.6%	32.0%

for each strategy when using ten threads. The percentages show the speed-up of each strategy compared to (a) the default strategy and (b) the slowest strategy.

We did not measure the `xFirst` strategies, as they were determined to perform poorly for IFDS. For example, `TargetsWithManySourcesFirst` took more than 900 seconds using a single thread, thereby performing more than 1100% worse than its counterpart, the best-performing `TargetsWithManySourcesLast` strategy.

The data shows that using a suitable scheduling strategy can have a significant impact on execution time. Considering the evaluated strategies, `TargetsWithManySourcesLast` is the best strategy for our IFDS analysis. It is 37.0% faster than the worst strategy presented here (excluding the above-mentioned poorly performing strategies) and 27.3% faster than the default. Relative standard deviations (RSD) for the reported measurements are between 4.0% and 10.4%. Figure 3 gives a graphical representation of the performance of the different scheduling strategies for different thread counts.

The effect of each strategy is application-dependent and may differ with the number of cells, the number of dependencies and cycles, and the costs of the used continuation functions. The advantage of `TargetsWithManySourcesLast` for the IFDS analysis can be explained by considering the aggregation of results; i.e., avoiding notifications of dependents. In the case of cells with many sources it pays off to hold back the update as long as possible, because this potentially allows aggregation with updates from other sources. Recall that before a continuation is actually invoked, the most current value(s) is(are) queried again and passed to the continuation in an aggregated form.

That way, the target cell can compute its result on a larger batch of information in one step as opposed to multiple small steps, which would be needed, if the cell was informed prematurely. This strategy works well for IFDS, because all propagations need to be handled in the same way and there are no special values, which would lead to early finalization of cells and could therefore be advantageous, as in the case of the purity analysis.

*Purity Analysis.* In addition to the analysis-independent strategies, we used the previously introduced `LatticeValueStrategy` adapted to the purity analysis. It makes use of the specific effect an update of a source cell may have on a target cell. If a dependee is *impure*, we can immediately decide on the purity of the depender and complete it with the value *impure*. The strategy gives such propagations a high priority as they lead to final results quicker. In contrast, if a cell is completed with *pure*, a target cell can *just kill* the dependency, because that update will not affect the current cell's value.

**Table 2: Runtimes and speed-ups for the Purity Analysis**

Speed-up shown compared to (a) default and (b) slowest strategy.

Strategy	Run time [s]	Speed-up (a)	Speed-up (b)
DefaultScheduling	0.42	0.00%	40.8%
SourcesWithManyTargetsFirst	0.70	-66.7%	1.41%
SourcesWithManyTargetsLast	0.70	-66.7%	1.41%
TargetsWithManySourcesFirst	0.71	-69.0%	0.00%
TargetsWithManySourcesLast	0.71	-69.0%	0.00%
TargetsWithManyTargetsFirst	0.69	-64.3%	2.82%
TargetsWithManyTargetsLast	0.68	-61.9%	4.23%
SourcesWithManySourcesFirst	0.68	-61.9%	4.23%
SourcesWithManySourcesLast	0.69	-64.3%	2.82%
LatticeValueStrategy	0.71	-69.0%	0.00%

Table 2 shows the results for all strategies using ten threads along with their relative speed-ups. The `DefaultStrategy` is significantly faster than the other strategies for this experiment. The other strategies, including the `LatticeValueStrategy` that prioritizes impure updates, show no significant differences in runtime. Relative standard deviations are between 1.0% and 6.1%. The `HandlerPool` uses a `java.util.concurrent.ThreadPoolExecutor` for pluggable scheduling strategies as this allows the necessary priority queue to be supplied. For `DefaultStrategy`, however, a `java.util.concurrent.ForkJoinPool` is used that already implements a work stealing LIFO queuing scheme. We believe that the `DefaultStrategy` is faster, because the continuation tasks created by the simple purity analysis are extremely small. This gives the default `ForkJoinPool` an advantage over the more complex `ThreadPoolExecutor` used for the other strategies.

The purity and IFDS analyses benefit from different strategies, emphasizing the need for pluggable, user-supplied strategies. While the analysis specific strategy did not benefit the simple purity analysis, it has been shown in the past [33] that such strategies can further improve performance.

### 3.3 RQ3: Scalability with Thread Count

To answer RQ3, we measured the speed-ups that RA2 achieves for different thread counts compared to the nearly identical solver that uses OPAL's single-threaded, but highly optimized fixed-point computations framework<sup>4</sup> [28]. The IFDS client analysis is identical. The goal is to evaluate the overhead and benefits of parallelization.

Figure 3 shows how the performance changes with the number of threads used for the IFDS analysis. Depending on the scheduling strategy, the speed-up with two threads compared to one thread is between 1.6x and 2.0x. Note that better strategies in general show lower speed-ups. When increasing the number of threads further, the speed-up increases up to 6.2x for 20 threads. With speed-ups of more than 4.8x for 10 threads, according to Amdahl's law, more than 88% of the execution is parallelized. Relative standard deviations for the reported measurements are between 1.8% and 13.7%.

The RA2-based implementation of the IFDS analysis is 28% slower than the sequential implementation in OPAL when a single

<sup>4</sup>Version 3.0.1-SNAPSHOT

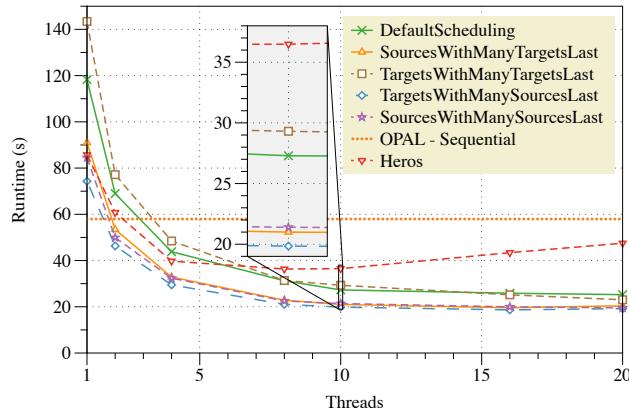


Figure 3: Performance with different numbers of threads

thread is used. The latter yielded runtimes of 58.0 seconds (RSD 8.5%), compared to 74.3 seconds for our best strategy. As expected, RA2 is slowed down by overhead related to enabling concurrency, which in this case is not needed. However RA2 clearly outperforms OPAL as soon as multiple threads are used. For the best strategy the speed-up over OPAL ranges between 1.3x with two threads, 2.9x with 10 threads, and 3.0x with 20 threads.

#### 3.4 RQ4: Comparison to Other IFDS Solvers

To answer RQ4, we compare our IFDS analysis to several other state-of-the-art systems.

First, we compare the performance and speed-ups achieved to Heros<sup>5</sup> [15]. We used Heros because it is parallelized and also independent of the static analysis framework. It is very mature, widely used and freely available. As we also compared to OPAL’s solver and as our analysis is based on OPAL’s three address code representation, we again used this representation as the basis for our analysis in Heros. By using the exact same base technology stack for both analyses, we ensure that we compare the raw performance of the solvers and that the results are not skewed by other technical differences. The IFDS client analysis was adapted to Heros’ interfaces, but performs the same analysis. Heros took 85.7 seconds on a single thread, 28% less than our DefaultScheduling, but already 15% more than our best strategy. For all thread counts measured, Heros had lower speed-ups than our best strategy (which is also the one with the lowest speed-ups), with a maximum of 2.36x at 8 threads. In comparison, our best strategy had a speed-up of 3.53 at 8 threads. Using more than 8 threads, Heros’ performance decreased significantly, while RA2’s performance increased until 16 threads (with a speed-up of 3.98x for the best strategy) and did not decrease significantly for 20 threads. Relative standard deviations for Heros were between 2.1% and 5.7%.

A direct empirical comparison with a parallelized implementation of IFDS using Actors [33] is unfortunately not possible. The solution was never publicly available and—according to the authors whom we contacted—is now practically impossible to get working again due to dependencies on unavailable and outdated beta

<sup>5</sup>Commit id: 46dda652

versions of libraries. However, they have also benchmarked their solution against a sequential implementation and we compare their speed-ups with our speed-ups against the sequential implementation using OPAL. The authors of IFDS-A reported a speed-up of 3.35x on 16 threads on eight cores compared to their own sequential implementation. Our implementation, on the other hand, using again 16 threads (on 10 cores), achieves a speed-up of 3.11x over OPAL’s highly optimized sequential implementation. Therefore, the performance of our implementation seems to be comparable to theirs, while our programming model is analysis-independent and, therefore, not specialized to IFDS. Among other things, the fact that two different baselines are used makes obvious that this comparison must be considered with caution and only as a work around the fact that a real comparison is not possible.

Finally, we evaluate against WALA 1.5.2 [39] as it provides another mature single-threaded IFDS implementation. As with Heros, we adapted the IFDS client analysis, this time with more effort because WALA’s IFDS solver is not framework independent. The analysis was, however, thoroughly checked to be equivalent to the one used with RA2 and OPAL. To overcome framework differences related to the underlying call graphs used by the different frameworks [31], we generated and serialized WALA’s RTA call graph using Judge [30] and deserialized it with OPAL. This ensures that the analyzed state space is equal. As WALA timed out after 10 hours when, analyzing the JDK, we performed a comparison with WALA on JavaCC 5.0. For this setup, WALA took 12.7 seconds (RSD 3.1%), while RA2, using the DefaultScheduling strategy, took 4.2 seconds on one thread (RSD 15.1%) and gave a speed-up of 4.5x (0.9 seconds, RSD 7.1%) using 16 threads. TargetsWithManySourcesLast, RA2’s best strategy, took 0.70 seconds on 16 threads (RSD 4.9%).

*Concluding remarks.* Overall, the experiments reported in this subsection clearly indicate that RA2 outperforms state-of-the-art parallel IFDS solvers. Compared with the state-of-the-art IFDS solver Heros, our implementation is faster on one thread, achieves higher speed-ups and scales to more threads. It also outperforms WALA’s IFDS solver significantly and seems to provide at least similar speed-ups as a specialized actor-based IFDS solver, despite being semi-implicit and analysis-independent.

## 4 THREATS TO VALIDITY

A threat to the validity of our claim about RA2 being able to provide semi-implicit parallelization of static analyses independent of the analysis, could be the evaluation of only two specific analyses, namely a purity analysis and an IFDS solver. We have tried to mitigate this threat by choosing two analyses as case studies that are fundamentally different in several ways. First, in their complexity, with the purity analysis being very simple and the IFDS solver being significantly more complex, but also in their use of the features provided by RA2’s programming model: while the purity analysis uses a singleton value lattice and the AggregationUpdater to perform joins automatically, the IFDS solver uses set-based properties and the MonotonicUpdater as joins are performed implicitly by the analysis. Additionally, the IFDS solver makes use of mutable state shared between the continuation invocations while the purity analysis does not. The evaluation also showed that these analyses benefit from pluggable scheduling strategies in different ways. Based on

this, we claim that our two studies span a wide range of analysis kinds.

A threat to the validity of our evaluation results is that a direct comparison was only possible to the parallel IFDS solver Heros, but not to the related IFDS-A solver, as its implementation is not available. We, however, compared our speed-ups against an equivalent sequential analysis in OPAL to those reported by the authors of IFDS-A. We also evaluated our analysis against another sequential IFDS solver from WALA, while another related IFDS solver from Flix is not working anymore and thus could not be compared.

## 5 RELATED WORK

*Parallel Static Analyses.* There exist several previous efforts to parallelize the solution of static analysis problems.

Heros [3, 15] is a parallel, state-of-the-art IFDS solver [32]; it is one of the benchmark implementations in our experimental evaluation (see Section 3). Later approaches, e.g., Boomerang [38] that built upon Heros, were not parallelized at all, however.

Méndez-Lojo et al. [26] parallelized a points-to analysis algorithm using the Galois system [17, 18] - a programming system for thread-safe parallel iteration over unordered sets. Like RA2, their approach relies on an underlying programming framework to provide thread-safety out-of-the-box to the analyses. Unlike the programming model underlying RA2, Galois is, however, a generic framework for concurrent programming over unordered sets and it is not specifically tailored to static analysis. For example, it does not provide support to automatically find fixed-points. As a result, the approach by Méndez-Lojo et al. is not directly applicable for the parallel execution of static analyses like RA2.

Rodriguez and Lhotak present IFDS-A [33], an algorithm for solving IFDS dataflow analysis problems using the actor model [1, 16] of concurrency. In order to apply IFDS-specific scheduling strategies, the authors were required to completely exchange the Scala Actors scheduler [12] with their own implementation. Combined with their custom strategy, this was necessary for significant performance improvements. The authors reported that IFDS-A outperforms their own equivalent sequential IFDS reference implementation with 4 or more cores with a speed-up of 3.35x using 16 threads. In contrast to IFDS-A, our approach is not limited to parallelizing IFDS, e.g., Section 3 evaluates a parallel purity analysis *not* based on IFDS. While being more general and using a scheduling strategy not specific to IFDS, our approach achieves a speed-up of up to 3.11x over an equivalent, optimized sequential implementation using 16 threads. At the same time, our approach outperforms sequential IFDS using 2 cores only (by 1.3x) instead of 4 cores as in the case of IFDS-A. Finally, our pluggable scheduling strategies enable significant performance improvements (see Section 3).

In the area of points-to analyses, Datalog has been used extensively as the underlying programming framework [4, 9, 40, 41]. Analyses are specified in terms of Datalog rules and executed using Datalog solvers. The use of parallel solvers enables automatic parallelization. Soufflé [37] is a parallel Datalog solver specifically developed for static analyses. It takes the analysis specification as an input and compiles it to a C++ program. Soufflé makes use of OpenMP, e.g., to parallelize nested join loops. Using 4 cores, a speed up of 2.1x compared to the sequential version was achieved.

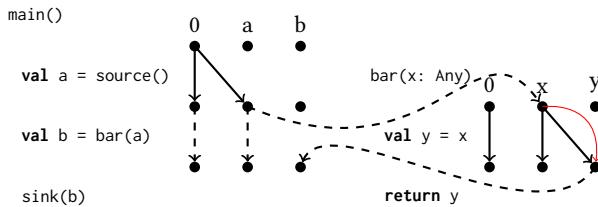
Similar speed-ups of over 2x have been reported for the Doop static analysis framework when using Soufflé as its underlying solver, but using 4 to 8 threads on 24 cores [2]. In contrast to RA2, these approaches only work on set-based lattices.

Flix [21] overcomes this issue—it is a declarative, rule-based language inspired by Datalog, capable of solving arbitrary fixed-point computations on lattices. Even though Flix is amenable to automatic parallelization due to its declarative nature, no parallel solver has been proposed for it yet. At some point in the past, Flix also provided a proof-of-concept implementation of IFDS, however, this is not working with the current version of Flix anymore.

*Reactive Frameworks for Static Analyses.* Reactive programming provides abstractions for event streams and time changing values (signals) [7, 22, 24, 27, 35, 36], which are well-suited for smart dependency management for static analyses. However, general-purpose reactive programming approaches cited above organize computations in an acyclic graph – by being general-purpose, they have no means to resolve cycles out-of-the box and hence the requirement that the graph is acyclic. However, cyclic data dependencies are essential for the target domain of static analysis. To address this need, the reactive programming framework underlying RA2 is more specialized. It is a reactive framework for time efficient, concurrent, fixed-point computation on lattices, which enables it to resolve cycles in a general way. We build on our previous work, Reactive Async [10], a programming system for deterministic concurrency in Scala that extends lattice-based shared variables, LVars [19], with cyclic data dependencies, which are resolved after the computation has reached a quiescent state and shared variables are no longer updated. RA2 significantly extends upon Reactive Async. First, RA2 allows shared variables (cells) to use a sequential update strategy, enabling continuations to safely access shared mutable state by executing them sequentially. Supporting shared mutable state is essential for implementing a state-of-the-art IFDS solver which is the basis for our experimental evaluation (see Section 3). Second, RA2 allows custom cell updaters such that monotonicity violations are detected dynamically while expensive additional joins can be omitted. Third, RA2 supports pluggable scheduling strategies which, as shown in Section 3, have a significant performance impact and allow analysis-specific tuning of the parallelization. Finally, aggregation of updates, both for a single dependee and across multiple dependees, reduces the number of continuation invocations for further performance improvements.

## 6 SUMMARY AND FUTURE WORK

In this paper, we proposed a new programming model for parallelizing static analyses based on the ideas and concepts of reactive programming, that (I) is semi-implicit, requiring only minor adaptions to analyses to benefit from parallelization, (II) is analysis-independent, lending itself very well towards the implementation of a wide range of static analyses, including purity and data-flow analyses. On top of our framework, which implements the proposed model, we implemented an IFDS solver and used the latter to implement a classical taint-flow analysis. The evaluation shows that our approach, while analysis-independent, is able to significantly outperform the highly-specialized, explicitly parallelized IFDS solver Heros, achieving both better performance and scalability.



**Figure 4: Simple IFDS taint analysis**

In future work, we plan to investigate whether on-the-fly profiling could be used by Reactive Async to automatically find the best scheduling strategy. Furthermore, we will investigate how to integrate speculative parallelization techniques to further increase the overall degree of parallelization. The presented programming model requires explicit continuations for processing updates of analysis results. By building on previous work on direct-style concurrency in Scala [11, 34], we plan to remove these explicit continuations, thereby making the programming model easier to use. Moreover, we would like to conduct user studies in order to evaluate the effect of our semi-implicit parallelization approach on program comprehension. Finally, our experimental evaluation suggests that the development of a standard benchmark suite for parallel program analyses will be needed in the future to ensure faithful comparisons between a growing number of approaches and frameworks.

## ACKNOWLEDGMENTS

This work was supported by the DFG as part of CRC 1119 CROSS-ING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

## APPENDIX

This appendix provides a short introduction to the IFDS analysis framework [32] for interprocedural, finite, distributive subset problems. IFDS is a dataflow analysis framework based on graph reachability that provides context- and flow-sensitivity. IFDS analyses are given by a finite domain of boolean facts that may or may not hold at a specific statement of the program. Four so-called flow-functions then need to be defined by users to define the effects of statements on the dataflow facts.

As shown in Figure 4, which depicts a simple taint analysis, the dataflow facts (here, whether a local variable is tainted) at each statement are represented by nodes. The edges are given by the flow-functions which describe the effects of non-call statements (normal flow, regular edges) and of call statements (call flow mapping facts from the caller's scope to the callee's scope, dashed edge from main to the beginning of bar; return flow mapping facts back to the caller's scope, dashed edge from the end of bar back to main; call-to-return flow for facts unaffected by the call, dashed edges inside main). The resulting graph is known as the exploded supergraph of the program as it is an expansion of the program's interprocedural control-flow graph. The special fact 0 represents the tautological

fact that holds everywhere. A fact is considered to hold at a program statement if it is reachable from the 0 fact. IFDS is efficient by reusing parts of the graph computed for one method by computing path edges (e.g., thin red edge in Figure 4) which summarize for a function whether a node is reachable given the reachability of a node at the method's start.

## REFERENCES

- [1] Gul A. Agha. 1990. *ACTORS - A Model of Concurrent Computation in Distributed Systems*. MIT Press.
- [2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: a Tale of Inter-Engine Portability for Datalog-Based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM, 25–30.
- [3] Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *SOAP*. 3–8.
- [4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. 243–262.
- [5] DoopBenchmarks [n.d.]. Doop Benchmarks. <https://bitbucket.org/yanniss/doop-benchmarks>.
- [6] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini. 2018. Lattice Based Modularization of Static Analyses. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA'18)*. ACM, 113–118.
- [7] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273.
- [8] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *TOPLAS* 23, 6 (Nov. 2001), 685–746.
- [9] Elnar Hajiyev, Mathieu Verbaere, and Oge De Moor. 2006. Codequest: Scalable Source Code Queries with Datalog. In *European Conference on Object-Oriented Programming (ECOOP'06)*. Springer, 2–27.
- [10] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: Expressive Deterministic Concurrency. In *SCALA@SPLASH*. ACM, 11–20.
- [11] Philipp Haller and Heather Miller. 2019. A Reduction Semantics for Direct-Style Asynchronous Observables. *J. Log. Algebr. Meth. Program.* 105 (2019), 75–111.
- [12] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science* 410, 2–3 (2009), 202–220.
- [13] Tim Harris and Satnam Singh. 2007. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*. Association for Computing Machinery, New York, NY, USA, 251–264.
- [14] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. 2018. A Unified Lattice Model and Framework for Purity Analyses. In *ASE*. ACM, 340–350.
- [15] Heros [n.d.]. Heros IFDS/IDE Solver. <https://github.com/Sable/heros>.
- [16] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*. Standford, CA, USA, August 20–23, 1973. William Kaufmann, 235–245.
- [17] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. 2009. How much Parallelism is there in Irregular Applications?. In *PPoPP*. ACM, 3–14.
- [18] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *PLDI*. ACM, 211–222.
- [19] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze After Writing: Quasi-Deterministic Parallel Programming with LVars. In *POPL*. ACM, 257–270.
- [20] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases. In *SIGSOFT FSE*. ACM, 98–108.
- [21] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *PLDI*. ACM, 194–208.
- [22] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (2018), 689–711.
- [23] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*. Association for Computing Machinery, New York, NY, USA, 65–78.

- [24] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, Article 11, 1 pages.
- [25] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-Based Points-to Analysis. In *PPoPP*. ACM, 107–116.
- [26] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-Based Points-to Analysis. In *OOPSLA*. ACM, 428–443.
- [27] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20.
- [28] Opal [n.d.]. OPAL. <https://github.com/stg-tud/opal>.
- [29] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary W. Hall. 2011. EigenCFA: Accelerating Flow Analysis with GPUs. In *POPL*. ACM, 511–522.
- [30] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 251–261.
- [31] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 107–112.
- [32] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM, 49–61.
- [33] Jonathan Rodriguez and Ondrej Lhoták. 2011. Actor-Based Parallel Dataflow Analysis. In *CC*. Springer, 179–197.
- [34] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional programming (ICFP'09)*. ACM, 317–328.
- [35] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*. ACM, New York, NY, USA, 25–36.
- [36] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. Association for Computing Machinery, New York, NY, USA, 796–807.
- [37] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 196–206.
- [38] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *ECOOP (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 22:1–22:26.
- [39] Wala [n.d.]. Wala. <http://wala.sourceforge.net>.
- [40] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.
- [41] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. Association for Computing Machinery, New York, NY, USA, 131–144.

# Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing

Yun Lin

National University of Singapore  
Singapore  
dcslyin@nus.edu.sg

Jun Sun

Singapore Management University  
Singapore  
junsun@smu.edu.sg

Gordon Fraser

University of Passau  
Germany  
gordon.fraser@uni-passau.de

Ziheng Xiu

National University of Singapore  
Singapore  
e0140856@u.nus.edu

Ting Liu

Xi'an Jiaotong University  
China  
tingliu@mail.xjtu.edu.cn

Jin Song Dong

National University of Singapore  
Singapore  
dcsdjs@nus.edu.sg

## ABSTRACT

In Search-based Software Testing (SBST), test generation is guided by fitness functions that estimate how close a test case is to reach an uncovered test goal (e.g., branch). A popular fitness function estimates how close conditional statements are to evaluating to true or false, i.e., the *branch distance*. However, when conditions read Boolean variables (e.g., `if(x && y)`), the branch distance provides no gradient for the search, since a Boolean can either be true or false. This *flag problem* can be addressed by transforming individual procedures such that Boolean flags are replaced with numeric comparisons that provide better guidance for the search. Unfortunately, defining a semantics-preserving transformation that is applicable in an *interprocedural* case, where Boolean flags are passed around as parameters and return values, is a daunting task. Thus, it is not yet supported by modern test generators.

This work is based on the insight that fitness gradients can be recovered by using runtime information: Given an uncovered interprocedural flag branch, our approach (1) calculates context-sensitive branch distance for all control flows potentially returning the required flag in the called method, and (2) recursively aggregates these distances into a continuous value. We implemented our approach on top of the EvoSuite framework for Java, and empirically compared it with state-of-the-art testability transformations on 807 non-trivial methods suffering from interprocedural flag problems, sampled from 150 open source Java projects. Our experiment demonstrates that our approach achieves higher coverage on the subject methods with statistical significance and acceptable runtime overheads.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Search-based software engineering.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397358>

## KEYWORDS

search-based, testing, testability, program analysis

### ACM Reference Format:

Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-Based Testing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397358>

## 1 INTRODUCTION

Search-based software testing (SBST) transforms the problem of generating test cases into an optimization problem, where tests are iteratively evolved by effective meta-heuristic search algorithms. To achieve this objective, the search algorithms are guided by fitness functions that estimate how close tests are to achieving test objectives such as branch coverage. Although SBST has been shown to be successful and effective in practice in many different domains [11, 14, 20, 28, 30, 33, 35, 41, 53, 58], in particular unit testing [26, 45], there is evidence that SBST does not achieve optimal coverage [7, 25], which negatively impacts the fault detection potential of generated test suites [47].

```
1 public int example(int a, int b){  
2     int x = a + b;  
3     if(Math.abs(x)==123){...}  
4     boolean y = a > b && a < b + 10;  
5     if(y==true){...} //flag problem  
6     boolean z = f(a, b)  
7     if(z==true){...} //interprocedural flag problem  
8 }
```

**Listing 1: Example method containing for a regular flag problem and an interprocedural flag problem.**

A primary challenge in SBST is defining an effective fitness function. A widely used and effective fitness function employed by many well-known SBST tools is based on *branch distance* [40]. A branching node  $n$  in a program can be regarded as an operator comparing two operands,  $op_1$  and  $op_2$ . Given a test case  $t$  exercising one branch of  $n$ , the branch distance from  $t$  to (covering) the other branch of  $n$  is defined by the value difference of  $op_1$  and  $op_2$ . For example, given the example program in Listing 1, the branch distance from a test case  $t \rightarrow a=-3, b=0$  to the true branch of the branching node at

line 3 is  $123 - | - 3| = 120$ , i.e., the corresponding ‘fitness’ for  $t$  with regard to the true branch is 120. The fitness function guides search algorithms towards reaching coverage objectives; for example, an optimal test case for the example branch is  $a=-100$ ,  $b=-23$  with a branch distance of 0, i.e., it covers the branch.

An important problem with this approach is the *flag problem* [13]: A flag problem is present when the branching condition reads boolean variables, e.g.,  $y==\text{true}$  (line 5 in Listing 1). In such a case, the branch distance is either 0 (e.g., when  $y=\text{true}$ ) or 1 (e.g., when  $y=\text{false}$ ). As a result, it is no longer a quantitative measure on how far a test case is from covering certain branch, and thus there is no *gradient* in the fitness landscape that the search algorithm could use to reach the objective of covering the branch. A commonly proposed approach to address the flag problem is testability transformation [10, 13, 29, 31, 34, 40]. The idea of testability transformation is to transform the intermediate representation of the source code so that (1) the transformed code preserves the semantics and (2) the boolean variables in branching conditions are replaced by predicates reading non-boolean variables. For instance, the condition  $y==\text{true}$  at line 5 of Listing 1 is replaced with  $a>b \ \&\ a<b+10$  after the transformation. Flag removal approaches use different strategies to transform different types of boolean flags [31]. For example, Baresel *et al.* [10] and Binkley *et al.* [13] proposed different flag removal approaches to resolve flags assigned in loops.

Existing work on flag removal only focuses on flags contained inside individual procedures. However, if branching conditions are based on method calls with boolean return-types, this problem becomes interprocedural. For example, consider Line 6 in Listing 1: The flag  $z$  is assigned a boolean return value, and the branch distance for the if-condition in line 7 can only either be 0 or 1. The interprocedural flag problem (IPF) is prevalent: In an empirical study on all methods from 150 open source Java projects we found that 11.8 % of them suffer from interprocedural flag problems (see Section 4.2 for more details). There currently is no satisfactory solution to the interprocedural flag problem. Although attempts have been made at defining transformations to recursively rewrite all boolean types in an object-oriented program [34], correct transformation rules that preserve semantics turn out to be too intricate to be practical. Consequently, the interprocedural flags remain a problem in SBST.

In this paper, we propose a lightweight recursive and context-sensitive approach to address the interprocedural flag problem. Unlike existing approaches based on testability transformation, our approach addresses the problem only through code *instrumentation* rather than transformation. Once we detect an uncovered program branch suffering from the interprocedural flag problem, we statically analyze the control flow graph of the function producing the boolean value and identify all potential program paths in the function which may return the flag. Then, we instrument these program paths so that we can dynamically calculate a fitness indicating how far a test case is away from exercising each path towards the required flag. Then, we aggregate these fitness values into a continuous value as the cumulative branch distance. Our approach is context-sensitive as it distinguishes method calls from different call sites or iterations of loops, and recursive as it handles cascading flag method calls in branching conditions. We conduct our experiment on 807 non-trivial Java methods with interprocedural flag problem,

```

1 Integer get(int[] keys, int value){
2     if(!checkPrecondition(keys, value))
3         return null;
4     for(int i=0; i<keys.length; i++)
5         if(checkValue(keys, i, value))
6             return keys[i];
7     return null;
8 }
```

**Listing 2:** Target method under test, containing two interprocedural flags.

```

1 boolean checkPrecondition(int[] keys, int value) {
2     if(checkArrayRange(keys)
3         && checkValueRange(value))
4         return true;
5     return false;
6 }
7
8 boolean checkArrayRange(int[] keys) {
9     return keys.length < 10000;
10 }
11
12 boolean checkValueRange(int value) {
13     return Math.abs(value) < 10000;
14 }
```

**Listing 3:** Precondition checking code, which is called by the target method and produces a boolean flag.

sampled from 150 open source Java projects. The experiment results demonstrate that our approach outperforms state-of-the-art approach (51.1% v.s. 47.9% coverage within same time budget) with statistical significance and acceptable runtime overheads.

In summary, this paper makes the following contributions:

- We propose a lightweight context-sensitive and recursive approach to address the interprocedural flag problem.
- We present an implementation of our technique based on *EvoSuite* [21], and the binaries and source code are made available at [1].
- We empirically show the prevalence of interprocedural flag problem in 150 open source Java projects. To the best of our knowledge, we are the first to show how common the interprocedural flag problem is in large-scale open source projects.
- We conduct an experiment on 807 methods suffering interprocedural flag problems sampled from 150 Java projects, showing the effectiveness of our approach.

The rest of the paper is structured as follows. Section 2 presents a motivating example. Section 3 describes our approach in detail. Section 4 evaluates the effectiveness of our approach. Section 5 reviews related work. Section 6 discusses related issues and concludes the paper.

## 2 MOTIVATING EXAMPLE

In this section, we motivate our approach using the example shown in Listing 2, which is a simplified version of the *OpenIntToField-HashMap.get()* method in the Apache Commons Math project [2].

The example function takes two input parameters, *keys* and *value*, and returns the corresponding key from the array *keys* if *keys* contains the computed key of *value*. It first checks the

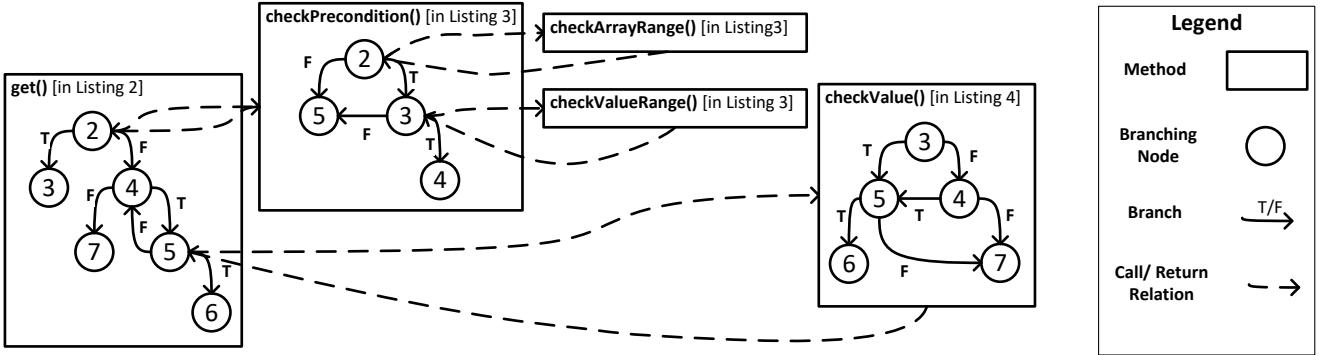


Figure 1: Program Dependency Graph for the example method in Listing 2.

```

1 boolean checkValue(int[] keys, int index, int val) {
2     int key = hashKey(val);
3     if(keys[index] == 0
4         || val >= Math.pow(2, index) + 100)
5         && keys[index] == key)
6     return true;
7     return false;
8 }
9
10 int hashKey(int key) {
11     final int h = key ^ ((key >> 20) ^ (key >> 12));
12     return h ^ (h >> 7) ^ (h >> 4) + 100;
13 }
```

#### Listing 4: Key comparison code, which is called by the target method and produces a boolean flag.

legitimacy of the inputs (line 2 in Listing 2) and iteratively goes through each element in keys to check whether the corresponding key of value is contained in keys (line 4–6). It returns null if either the inputs does not satisfy the precondition (line 3) or the key of value is not contained in keys (line 7). The program contains two method calls in line 2 and 5. The details of the invoked methods are shown in Listing 3 and 4.

For readability, we show the program dependency graph in Figure 1, where each method is represented as a rectangle. Within a rectangle, the control flow graph (CFG) of the method is shown where each node is represented as a circle labelled with its source code line number and the control flows between nodes are represented by lines labelled by branch value (i.e., true or false). Connecting the rectangles, the call relation is represented by dashed lines. For simplicity, we skip the CFG of `checkArrayRange()` and `checkValueRange()`.

Automatically generating test cases for method `get()` to achieve high branch coverage is non-trivial, in particular for the following two branches:

- The branch  $\langle 2, 3 \rangle$  in line 2 (i.e., the edge  $\langle 2, 3 \rangle$ , Listing 2).
- The branch  $\langle 5, 6 \rangle$  in line 5 (i.e., the edge  $\langle 5, 6 \rangle$ , Listing 2).

Using random testing, the branch  $\langle 2, 3 \rangle$  is hard to cover because randomly generated test cases are most likely to violate the precondition (see Listing 3)<sup>1</sup>. The second branch is hard to cover because

<sup>1</sup>Theoretically, the sampling space should range from  $2^{-31}$  to  $2^{31} - 1$ . Nevertheless, existing tools such as *EvoSuite* usually sample with bias towards a smaller range such as  $[-1024, 1024]$ .

the mapping rule from a value to its key is complicated (see line 2 in Listing 4) and thus it is hard to randomly generate a pair of “magical” key (in `keys`) and corresponding value under the guarding conditions (see the condition in line 5 in Listing 4).

Testing method `get()` using SBST suffers from the interprocedural flag problem. For instance, the state-of-the-art SBST tool *EvoSuite* is unable to generate tests with sufficient branch coverage. This is because *EvoSuite* uses branch distance to quantitatively measure how far a test suite is from the uncovered branches. In this example, a human reader can observe that the test case  $t \leftarrow \text{keys}=[0]$ ,  $\text{value}=999$  is a “closer” candidate towards covering branch  $\langle 2, 3 \rangle$  in Listing 2 than  $t' \leftarrow \text{keys}=[0]$ ,  $\text{value}=99$ . However, existing branch distance (e.g., [40]) is defined based on the operands of the branching nodes (e.g., in line 2, based on the returned boolean values from `checkPrecondition()`). If the operands are boolean variables, the branch distance will always be evaluated to 0 or 1 and thus do not provide a measurement on how close a test suite is to cover certain branch.

The key to address such an interprocedural flag problem is to aggregate the branch distances in the called methods into one combined value to recover the “gradient” towards the uncovered branch. The following three challenges make this a non-trivial problem:

**C1: The multiple paths problem.** The called method may have multiple program paths, each of which returns a boolean value. For example, for covering the branch  $\langle 2, 3 \rangle$  in method `get()`, we require a test case to exercise line 5 in Listing 3. However, there are two paths leading to node 5, i.e., path  $\langle 2, 5 \rangle$  and path  $\langle 2, 3, 5 \rangle$  in method `checkPrecondition()`. Note, that exercising path  $\langle 2, 5 \rangle$  requires creating an array of length of 10,000, while exercising path  $\langle 2, 3, 5 \rangle$  only requires generating a large value. Moreover, during test suite optimization, the likelihood of exercising either path sometimes varies from generation to generation (if we use a genetic algorithm). Consequently, how can we take multiple paths into consideration and aggregate their branch distances?

**C2: The cascading interprocedural flag problem.** The called method may suffer from further interprocedural flag problems. We can observe that methods with boolean return types are invoked at line 2 and line 3 in Listing 3. In general, such interprocedural flag problems may cascade for multiple layers (depending on the call

graph depth) or infinitely many layers (e.g., recursive method calls). How do we aggregate the branch distance then?

**C3: The call and iteration context problem.** The branch distance of a branching condition of a boolean method call is sensitive to the call site and loop iteration context. For example, exercising branch  $\langle 5, 6 \rangle$  requires that `checkValue()` returns true (i.e., exercising line 6 in Listing 4). However, a test case may exercise different paths during different iterations of the loop. For instance, the test case  $t \leftarrow \text{keys}=[0, 1], \text{value}=0$  calls `checkValue()` twice. The first one exercises path  $\langle 3, 5, 7 \rangle$  and the second one exercises path  $\langle 3, 4, 7 \rangle$ . How do we aggregate the branch distances of different iterations?

Note that the above problems may co-occur and happen in a cascading way. For example, there may be a chain of boolean-typed method calls; each called method may be invoked in a loop; and each method may have multiple paths to return a boolean value. In this work, we propose a recursive and context-sensitive approach to address the above problems systematically.

### 3 APPROACH

#### 3.1 Definitions

We first define the terminology. We call the method under test the **target method**, denoted as  $m_t$ . We refer to a method  $m$  as a called method if  $m$  is called directly in the target method  $m_t$  or indirectly through a called method. A called method  $m$  is called a **flag method** if there exists a condition  $c$  of a branching node in the target method  $m_t$  which directly or indirectly depends on the value returned by  $m$ . In addition, we call *that* boolean value used in the condition  $c$  an **interprocedural flag** and the branch to be covered an **interprocedural flag branch**. For example, method `checkPrecondition()` in Listing 2 is a flag method of the target method `get()`. Given a test case  $t \leftarrow \text{keys}=[0, 1], \text{value}=-20$ , there are two uncovered interprocedural flag branches in `get()` method, i.e., branch  $\langle 2, 3 \rangle$  and branch  $\langle 5, 6 \rangle$ . The corresponding interprocedural flags are the values returned by method `checkPrecondition()` and by method `checkValue()`.

Furthermore, given an interprocedural flag branch  $b$  which depends on a flag method  $m$ , there may be multiple program paths in  $m$  which return a boolean value such that  $b$  is covered. We call such program paths in  $m$  as the **required paths** for covering  $b$ . We call the last branch in a required path as the **required branch**. For example, for branch  $\langle 2, 3 \rangle$  in Listing 2, the path  $\langle 2, 3, 4 \rangle$  in Listing 3 is a required path, and the branch  $\langle 3, 4 \rangle$  is a required branch.

#### 3.2 Overview

The goal of our approach is to recover the missing quantitative measurement on how far a test case is from covering a branch due to the interprocedural flag problem. Our rationale is that, given an interprocedural flag branch, we *collect and aggregate the branch distances of the required paths in the called flag methods so that we can have one quantitative measurement*. Figure 2 shows an overview of our approach, where each ellipse represents a process and each rounded rectangle represents an artifact. Grey rectangles represent input and output while grey processes represent our key contributions. Our approach takes as input a target method and a test case

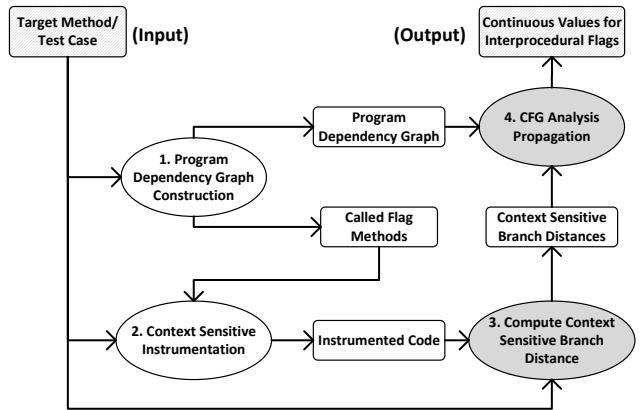


Figure 2: Approach Overview

(which can be generated automatically) and generates quantitative branch distance for evaluating how far the test case is from every interprocedural flag. Given the target method, we first construct its program dependency graph (e.g., the one in Fig. 1) so that we identify all the called methods and their control flow graphs (**Step 1**). Next, we instrument the target method and its called methods in order to obtain the required branch distances at runtime. Note that the instrumentation is sensitive to call context and iteration context (**Step 2**). Then, we run the test case for the target method and collect branch distances based on the required paths for every interprocedural flag (**Step 3**). Finally, we recursively analyze the CFG of the relevant called methods to aggregate the branch distances based on the required paths to a branch distance for every interprocedural flag in the target method (**Step 4**).

As mentioned in Section 2, our approach must address three challenges, i.e., the multiple path problem, the cascading interprocedural flag problem, and the call and iteration context problem. Next, we introduce (1) how we distinguish different call/iteration context, (2) how we aggregate branch distances in called flag methods, and finalize the section with (3) the overall fitness calculation algorithm.

#### 3.3 Context Sensitive Branch Distance

In the following, we first introduce how branch distance are computed at runtime in classical approaches with context-insensitive branch distance, and how our approach addresses their limitations.

**Branch Distance.** Suppose that the target method  $m_t$  has a branching node  $n$ . Assume that a test case  $t$  of  $m_t$  exercises  $n$  precisely once. The branch distance of a branch  $b$  of  $n$  with regard to  $t$  is written as  $distance(b, t)$  where the condition in  $b$  is to be satisfied in order to exercise branch  $b$ . Let us denote  $b.c$  as the condition in  $b$ , function  $distance(b, t)$  is defined as follows [40]:

```

1 boolean example(int x) {
2     if(isFlag(x)) x++;
3     if(isFlag(Math.pow(2, x))) return true;
4     return false;
5 }
6
7 boolean isFlag(int x){
8     if (x < 1)
9         return true;
10    return false;
11 }

```

**Listing 5: Example for Context Sensitive Branch Distance**

$$distance(b, t) = \begin{cases} 0 & \text{if } b.c \text{ evaluates to true} \\ K & \text{else if } b.c \text{ is false} \\ |a - b| + K & \text{else if } b.c \text{ is } a == b \\ K & \text{else if } b.c \text{ is } a != b \\ a - b + K & \text{else if } b.c \text{ is } a < b \\ a - b + K & \text{else if } b.c \text{ is } a <= b \\ b - a + K & \text{else if } b.c \text{ is } a > b \\ b - a + K & \text{else if } b.c \text{ is } a >= b \end{cases}$$

where  $K$  is a positive constant value which is the minimum value of all possible branch distances. In *EvoSuite*,  $K$  is set to be 1. Note that the branch distance is always a non-negative value. For instance, given the test case  $t' \rightarrow a=-20$ ,  $b=-3$ , the branch distance for the branch at line 3 in Listing 1 (i.e.,  $\text{Math.abs}(a+b) == 123$ ) is  $|-20 - 3| - 123| + K = 100 + K$  for the then-branch and 0 for the else-branch.

**Normalized Branch Distance.** If the test  $t$  does not cover the branching node  $n$  of the branch  $b$ , let  $n_p$  be the nearest branching node which is covered by  $t$  such that  $n$  control-depends on  $n_p$ , a normalized branch distance is used to quantify how far  $t$  is from covering  $b$ , as follows.

$$distance_n(b, t) = approach\_level + norm(distance_{n_p}(b', t)). \quad (1)$$

where *approach\_level* measures the number of branching node from node  $n_p$  to node  $n$  along the program path; *norm* is a function which normalizes the distance to a value between 0 and 1; and  $distance_{n_p}(b', t)$  is the branch distance defined above for the branch from  $n_p$  to  $n$  (i.e.,  $b'$  in Equation 1). The normalization function may be implemented differently in different SBST approaches. One candidate implementation is  $norm(x) = \frac{x}{1+x}$  [8]. For example, the test case  $t \rightarrow \text{keys}=[], b=\text{Integer.MAX\_VALUE}$  exercises branch  $\langle 2, 3 \rangle$  in *get()* method in Fig. 1. Thus, the approach level for branching node 5 in *get()* is 2 and  $distance_2(b, t)$  is 1, thus the overall  $distance_5(\langle 2, 3 \rangle, t) = 2 + \frac{1}{1+1} = 2.5$ .

**Context Insensitive Branch Distance.** A test,  $t$ , can exercise a branch,  $b$ , multiple times under different contexts. This can happen when the branching node of  $b$  is either (1) a loop node exercised for multiple iterations or (2) the method defining  $b$  is called in multiple sites. Listing 5 shows the example. The branching node at line 8 is executed twice given any test case for method *example()*. Assume that the branching node of a branch,  $b$ , is exercised  $m$  times during the execution of  $t$ . Let us use  $distance(b, t)_i$  for denoting the branch distance calculated for the  $i$ th time  $b$ 's branching node is

**Table 1: Sensitive Branch Distances for Iterations**

Iter	Passed Conditions	App Level	Branch Distance	Res
1	<3, 5>: key[index]==0 && <5, 7>: !(keys[index]==key)	0	100/101=0.99	0.99
2	<3, 4>: !(key[index]==0) && <4, 7>: !(val>=Math.pow(2, index)+100)	1	1/2=0.5	1.5

exercised. Existing state-of-the-art SBST tools like *EvoSuite* compute the branch distance of a branch  $b$  as follows:

$$distance(b, t) = \min\{distance(b, t)_i\}. \quad (2)$$

Note that,  $i \in [1, m]$  in Equation 2. With Equation 2, classical search algorithm records the minimum branch distance of a branch in the execution of  $t$ . While efficient in general, the information loss (by keeping only the minimum branch distance) incurs two limitations, which impairs the analysis of *interprocedural flag problem*.

**Limitation 1: Aggregation Confusion.** When an interprocedural flag problem happens (i.e., the classical algorithm is trapped to exercise an interprocedural flag branch), our rationale is to aggregate the branch distances in the called flag method to recover the gradient on the flag branch. However, if the flag method is called in different call sites of the target method, *there could be more than two branch distances collected for a single branch*, thus taking the minimum branch distance as in Equation 2 will convey misleading branch distance for at least one flag branch. Even worse, it may cause contradictory analysis result. For example, in Listing 5, if we are to cover the branches in line 2 (i.e., *isFlag(x)*) and line 3 (i.e., *isFlag(Math.pow(2, x))*), we should analyze the branch distance for the branch in line 8 (i.e.,  $x < 1$ ). Intuitively, the gradient of branch distance in this case can be recovered by simply using the branch distance in line 8. However, a test case  $t \leftarrow x=0$  causes a problem of confusing distance. The condition of first interprocedural flag branch (line 2) is evaluated to be true while that of the second is evaluated to be false. When we are to aggregate the branch distance in the true branch in line 8 (i.e.,  $x < 1$ ) back to the second interprocedural flag, Equation 2 shows that the branch distance is 0. The reason is that the branch in line 8 has been exercised, thus the minimum distance is 0. The result is contradictory to the fact that the second interprocedural flag has not yet been evaluated to true on  $t \leftarrow x=0$ .

**Limitation 2: Local Optima Effect.** Even if the call site of a flag method is unique in target method, taking the minimum branch distance may cause the search process to be stuck in a local optimum. When the flag method is called within a loop, each loop iteration derives a branch distance for the required branch in the called flag method. Evolving the test case with regard to only minimum branch distance pays a price of losing other optimizing opportunities. Taking example of the interprocedural flag branch  $b = \langle 5, 6 \rangle$  in Listing 4, the test case  $t \leftarrow \text{keys}=[0, 1], \text{value}=0$  for *get()* method call the *checkValue()* method twice (i.e., 2 iterations). For readability, we show the call graph in Figure 3, we show the target branch in *get()* method and the required branch in *checkValue()*

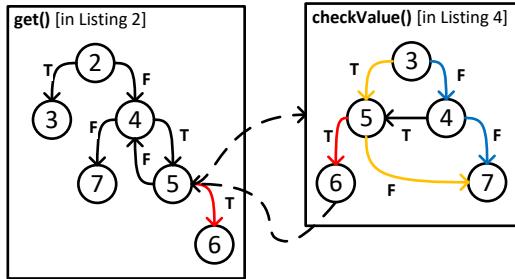


Figure 3: Local Optima Example

in red. The first iteration exercises the branches  $\langle 3, 5 \rangle$  and  $\langle 5, 7 \rangle$  in Listing 4 (yellow path in Figure 3), and the branch distance for  $b$  is 0.99. The second iteration exercises the branches  $\langle 3, 4 \rangle$  and  $\langle 4, 7 \rangle$  in Listing 4 (blue path in Figure 3). Table 1 shows the branching condition evaluated in each iteration and the value of branch distances. Note that, there are two branch distances for the uncovered branch  $\langle 5, 6 \rangle$ , as the path  $\langle 3, 4, 7 \rangle$  can be diverted from either node 3 or node 4. Their branch distances are 1.5 and 1.99 respectively.

According to Equation 2, state-of-the-art approach uses the branch distance in the first iteration (i.e., 0.99) as the branch distance for branch  $\langle 5, 6 \rangle$ . Intuitively, it favors diverting from yellow path over blue path to the target (red) branch. However, in this case, diverting from blue path is more feasible than from yellow path. The condition  $\text{keys}[index]==\text{key}$  (on branch  $\langle 5, 6 \rangle$ ) is extremely hard to satisfy when  $\text{keys}[index]==0$  (on branch  $\langle 3, 5 \rangle$ ) as the hash result key is usually a non-zero value (line 2 in Listing 4). In contrast, the path condition  $\text{value}>=\text{Math.pow}(2, \text{index}) \wedge \text{keys}[index]==\text{key}$  (on branches  $\langle 4, 5 \rangle, \langle 5, 6 \rangle$ ) is much easier to satisfy. For example, we can repetitively mutate the value of  $\text{value}$  and  $\text{keys}[index]$ . Nevertheless, given that we keep  $\text{distance}(\langle 5, 6 \rangle, t)_1$  as the branch distance, the branch distance is insensitive to changes of  $\text{value}$  and  $\text{keys}[index]$ . In this regard, the search algorithm will soon be stuck in local optima, leaving the fitness guidance ineffective.

**Solution.** Our remedy is to maintain the branch distance for a branch each time the branch is executed under different contexts and aggregate the branch distances (see details in Section 3.5). Figure 4 shows the meta-model for our instrumentation for distinguishing branch distance under various contexts. Each interprocedural flag branch has multiple required branches (see definition in Section 3.1). Each required branch has multiple iteration contexts, each of which is associated with a branch distance. An iteration context consists of a call context and the runtime execution trace towards the required branch. A call context is a call stack, each of its elements describes what method is called and the position (or specific instruction) to trigger the call. Moreover, we keep the runtime execution trace towards the required branch in the top call in the call stack so that we can further distinguish branch distance for various iterations by the execution trace in the call frame.

For example, given the test case  $t \leftarrow \text{keys}=[0, 1], \text{value}=0$ , we obtain a branch distance for the branch  $\langle 4, 5 \rangle$  for each iteration. Both

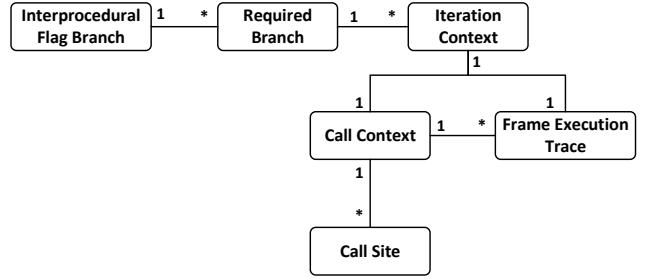


Figure 4: Meta Model for Branch Distance

distances share the same call context, i.e.,  $\text{checkValue}() \sim \text{get}()$ . In addition, we distinguish their iteration contexts as the first iteration has branch trace in  $\text{get}()$  as  $\langle 2, 4, 5 \rangle$  while the second iteration has branch trace as  $\langle 2, 4, 5, 4, 5 \rangle$ .

### 3.4 Graph Walking Algorithm

Given a test case and an uncovered interprocedural flag branch, we calculate its interprocedural branch distance by statically *walking through* the program dependency graph, starting from the interprocedural flag branch and ending by every of its required branch. For each required branch  $b$ , we compare the static walking trajectory (on program dependency graph) reaching  $b$  and all its iteration contexts to select valid branch distances. Then, we recursively apply two-stage aggregation to derive an interprocedural branch distance. In general, we first aggregate the branch distance for each required branch. Then we aggregate the interprocedural branch distance from aggregated distance of multiple required branches. The process is recursive because the required branch may still be an interprocedural flag branch. By this means, our approach first identifies the relevant branches (and their branch distance) by walking towards each required branch, and aggregate the branch distances by walking back towards the target interprocedural branch. Algorithm 1 and 2 show the details.

**Overall Computation.** Algorithm 1 takes as input an interprocedural flag branch  $b_{if}$ , its static walking call context  $wcc$ , and a table of context-sensitive branch distances collected by running a test case,  $rct$  (see Figure 4). It outputs an aggregated branch distance. The walking call context  $wcc$  is initialized as the target method. It grows each time we proceed to analyze a new called interprocedural flag method. The runtime context table  $rct$  maintains the context-sensitive branch distance for all the branches during the execution of test case.

When analyzing an uncovered interprocedural flag branch, we first parse the CFG of the flag method and identify all required branches (line 1). Then, we retrieve all the runtime execution traces towards those required branches under such a call context (line 2). As showed in Figure 4, a runtime execution trace consists of a sequence of branches in the CFG of the flag method. Technically, we maintain a map, mapping from a trace towards a required branch to a branch distance. Then, for each required flag branch, we compute its aggregated distance from all its execution traces (line 6–9). We will explain its details (i.e.,  $\text{branch\_fitness}_r()$  method) in Section 3.5.

**Algorithm 1:** Calculate Fitness for Interprocedural Flag Branch ( $branch\_fitness_{if}$ )

---

**Input** : an interprocedural flag branch  $b_{if}$ , walking call context  $wcc$ , runtime context table  $rct$   
**Output**: an aggregated branch distance  $fitness$

```

1  $branch\_set_r \leftarrow$ 
  identify_required_branches( $b_{if}.called\_cfg$ );
2  $id\_map \leftarrow rct.find(b_{if}, wcc)$ ; // return a map from iteration
  to a branch distance.
3  $fitness\_set \leftarrow \emptyset$ ;
4 for  $b_r \in branch\_set_r$  do
5    $fitness\_set_{iter} \leftarrow \emptyset$ ;
6   for  $trace \in id\_map.keys$  do
7      $fitness_{iter} \leftarrow branch\_fitness_r(b_r, wcc, trace, rct)$ ;
8      $fitness\_set_{iter} \leftarrow fitness\_set_{iter} \cup \{fitness_{iter}\}$ ;
9    $fitness \leftarrow$  aggregate  $fitness\_set_{iter}$  using Equation 3;
10   $fitness\_set \leftarrow fitness$ ;
11  $fitness_{agg} \leftarrow$  aggregate  $fitness\_set$  using Equation 3;
12 return normalize( $fitness_{agg}$ );
```

---

Afterwards, we can further compute an overall branch distance from aggregated branch distance of all the required branches (line 4–11). We will elaborate the aggregation details (e.g., Equation 3) in Section 3.5. Finally, we return the normalized fitness value.

For example, if we are to compute the interprocedural flag branch distance for branch  $\langle 5, 6 \rangle$  in `get()` method in Listing 2, with the test case  $t \leftarrow \text{keys}=[0, 1]$ ,  $\text{value}=0$ , Algorithm 1 first identifies that the required branch is branch  $\langle 5, 6 \rangle$  in `checkValue()` method. In addition, there are two iterations to call the `checkValue()` method. The corresponding branch traces are  $\langle 2, 4, 5 \rangle$  and  $\langle 2, 4, 5, 4, 5 \rangle$  in `get()` method respectively. Under the call context of `get():52`, for each required branch we maintain a map from branch trace to branch distance. For example, for the required branch  $\langle 3, 6 \rangle$  in `checkValue()` method, we maintain a map such as  $\langle \langle 2, 4, 5 \rangle = 0.99, \langle 2, 4, 5, 4, 5 \rangle = 1.99 \rangle$ . Then, we aggregate those branch distance accordingly. The details are as follows.

**Computation for Required Branch.** Algorithm 2 shows how we calculate branch distance for individual required flag branch. It handles various scenarios for calculating the branch distance, including when to aggregate the branch distance, and when to recursively call Algorithm 1 (i.e., `fitnessif()` in line 3 and 6 in Algorithm 2).

In general, Algorithm 2 handles three scenarios: (1) when the required branch is exercised (e.g., `isFlag(int x){return isFlag2(x);}`), in this case, we return the branch distance from reversing result of `isFlag2()`; (2) when the required branch is, again, an interprocedural flag branch  $b_r$  but not exercised (line 6), in this case, we return the branch distance from generating the same flag with  $b_r$ ; and (3) when the required branch has some non-flag branch distance (line 9). For the first and second scenarios, we update the call context and recursively call Algorithm 1. For the third scenario, we compute the branch distance from runtime context table as classical approach and recover the gradient.

<sup>2</sup>Here, we use line number 5 as the call site. Nevertheless, we use the bytecode instruction index in our implementation.

**Algorithm 2:** Calculate Fitness for Required Branch ( $branch\_fitness_r$ )

---

**Input** : a required flag branch  $b_r$ , walking call context  $wcc$ , iteration  $trace$ , runtime context table  $rct$   
**Output**: fitness  $f$

```

// the branch requires reversing the output of an exercised
interprocedural flag method.
1 if  $b_r$  is interprocedural flag branch &&  $b_r$  is exercised then
2    $wcc \leftarrow wcc.append(b_r.callsite)$ ;
3   return  $branch\_fitness_{if}(-b_r, wcc, rct)$ ;
// the branch requires exercising an interprocedural flag method.
4 if  $b_r$  is interprocedural flag &&  $b_r$  is not exercised then
5    $wcc \leftarrow wcc.append(b_r.callsite)$ ;
6    $fit \leftarrow branch\_fitness_{if}(b_r, wcc, rct)$ ;
7 else
8    $id\_map_r \leftarrow rct.find(b_r, wcc)$ ;
9    $fit \leftarrow id\_map_r.get(trace)$ ;
10 return  $fit$ ;
```

---

### 3.5 Aggregating Branch Distance

We design the aggregation of multiple branch distance based on the rationale that *we still favor the smallest distance, but the aggregated branch distance should also be sensitive to changes of other branch distance*. With that property, we can avoid the second limitation of Equation 2, i.e., stuck in local optima.

Let  $t$  be a test case and  $b$  be a branch with  $k$  branch distance, Let the  $i$ th branch distance be  $distance(b, t)_i$  ( $i > 0$ ), we aggregate the branch distance as follows:

$$distance(b, t) = \frac{k}{\sum_{i=1}^k \frac{1}{distance(b, t)_i}} \quad (3)$$

Equation 3 has the following properties:

- $distance(b, t)$  is equal to 0 if  $\exists i \in [1, k]$  so that  $distance(b, t)_i$  is 0.
- $distance(b, t) \rightsquigarrow 0$  if  $\exists i \in [1, k]$  so that  $distance(b, t)_i \rightsquigarrow 0$ .

The notation  $a \rightsquigarrow b$  stands for  $a$  is approaching the value of  $b$ . For the second property, we have that:

$$\frac{\partial distance(b, t)}{\partial distance(b, t)_i} = k \cdot \left( \sum_{i=1}^k \frac{1}{distance(b, t)_i} \right)^{-2} \cdot \frac{1}{distance(b, t)_i^2} \quad (4)$$

It means that, once  $\exists i$  so that  $distance(b, t)_i$  is the smallest distance, the derivative of  $distance(b, t)$  over  $distance(b, t)_i$  decreases fastest so that we can favor it to decrease to 0. By this means, we achieve the goal of favoring the smallest branch distance.

On the other hand, all the other branch distances are considered so that we can avoid being stuck in a local optima. For example, for branch  $\langle 5, 6 \rangle$  in Listing 4 (or Rectangle `checkValue()` in Fig. 1), the test case  $t \leftarrow \text{keys}=[0, 1]$ ,  $\text{value}=0$  has  $distance(\langle 5, 6 \rangle)_1 = 0.99$  while  $distance(\langle 5, 6 \rangle)_2 = 1.5$ . Therefore, when  $distance(\langle 5, 6 \rangle)_1$  stuck on local optima, the fitness is still sensitive to  $distance(\langle 5, 6 \rangle)_2$  and gradually evolves to a test case covering the target branch.

Last but not least, the numerator  $k$  is used to avoid test cases incurring a large number of iterations. Note that, if we let numerator be 1, the more loop iterations a test case incurs, the larger the denominator. As a result, the search algorithm will favor those test

cases incurring large number iterations. This phenomenon causes two drawbacks: (1) the number of iterations is usually irrelevant to exercise an uncovered branch; and (2) the test cases incurring many iterations have larger runtime overhead for the search process. In this regard, we add  $k$  as a penalizing factor to mitigate the problem.

## 4 EVALUATION

We build our proof-of-concept tool *Evoif* (*EvoSuite* on Interprocedural Flag) based on *EvoSuite* [21]. In the implementation of *Evoif*, we enhance *EvoSuite*'s instrumentation to support our meta model in Figure 4 and extend a new test case fitness function `fbranch` based on *EvoSuite* framework. The tool and its source code, as well as all the experimental data, are accessible at [1]. In this section, we aim to answer the following research questions:

- **RQ1:** How prevalent is the interprocedural flag problem in practice?
- **RQ2:** Does *Evoif* improve the testing performance on code suffering from interprocedural flag problems compared to the state-of-the-art approach?
- **RQ3:** Does *Evoif* incur acceptable overhead when code under test does not suffer from interprocedural flag problems?

### 4.1 Experiment Setup

**4.1.1 Benchmark Tool.** We choose *EvoSuite* [21] as our benchmark tool. *EvoSuite* supports various testability transformations, including transforming common methods in the Java `String` and `Collection` classes. For example, *EvoSuite* can transform the `String.equals()` method into one returning an `int` (based on the edit distance) to address the flag problem. It also supports a number of useful heuristics to cover challenging branches [21, 23].

**4.1.2 Subject Methods.** For this experiment, we use 807 methods suffering from interprocedural flag methods from 150 projects. The 150 projects consist of standard SF100 data set [22] and 50 extra complemented projects (we will discuss why we complement 50 projects later). Our selection (or method filtering) heuristics are fixed and the selection of those methods is replicable in our experiment. The heuristics are designed for *EvoSuite*'s limitation of class instrumentality and mutation capability. As for class instrumentality, current *EvoSuite* implementation cannot allow us to instrument JDK library classes. As a result, we miss the branch distance in some called flag method such as the `equals()` method in `java.lang.Object`. As for mutation capability, *EvoSuite* has limitation to instantiate legitimate complex parameter object (including abstract class and interface), e.g., `java.sql.ResultSet`. As a result, the *EvoSuite* framework keeps generating test cases triggering program crashes before executing the target methods during the evolution. In both cases, we cannot evaluate the performance difference between *EvoSuite* and *Evoif* from the perspective of code implementation.

**Filtering Heuristics.** Addressing the mutation capability is beyond the scope of this work. Therefore, to evaluate the effectiveness of our gradient recovering approach, we conservatively define the filtering rules (in Table 2) to select methods which are less likely to be affected by the limitation of mutation capability. In Table 2, we

**Table 2: Heuristics for Filtering Experimental Methods**

Categories	Description	#IPF
<b>H1</b>	Called IPF is not instrumentable.	1673
<b>H2</b>	Target method with no primitive parameters.	4573
<b>H3</b>	Target method with parameter of interface or abstract class type.	1468
<b>H4</b>	Called IPF method with no primitive parameters.	131
<b>H5</b>	Called IPF method with parameters of interface or abstract class type.	654
<b>Used IPF Methods</b>	The methods used for the experiment.	<b>807</b>
<b>Total</b>	/	9306

also list the quantity of methods filtered by every filtering heuristic. We will thoroughly discuss the filtering heuristics in details in Section 4.5 and Section 4.6.

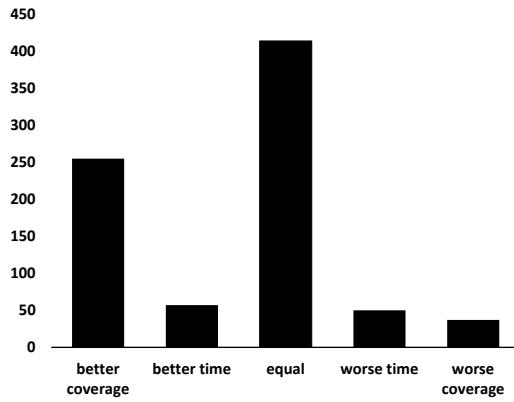
**Project Complement.** The heuristics in Table 2 cause the number of usable IPF methods from the traditional SF100 data set to be small. On one hand, some project in SF100, such as *greencow* (the 28th project in SF100), contains only one class with two branchless methods. In addition, 56% of the Java classes in SF100 benchmark contain only branchless methods [44]. On the other hand, our heuristics to get rid of complicated data structure remove a lot of methods (we will discuss it in Section 4.5 and 4.6). Therefore, following the convention in software testing community [44], we complemented 50 large popular Java open source projects to scale up our experiment. These projects are chosen considering their scale and popularity (e.g., *Weka* (v3.8.0) [5], *JFreeChart* (v1.5.0) [4], etc.). Some have been used in the annual unit test generation contest [46]. We deem them to be representative and realistic.

**4.1.3 Performance Evaluation.** In this experiment, both *Evoif* and *EvoSuite* stop when the 100s time budget is used up or they have achieved 100% branch coverage. Given that the used time and coverage of *EvoSuite* and *Evoif* on each target method is evaluated for 10 times, we compare the mean and median coverage and time respectively. We classify the performance comparison on a target method into 5 categories.

- **Better Coverage:** Within the time budget, *Evoif* has a better coverage than *EvoSuite*.
- **Worse Coverage:** Within the time budget, *Evoif* has a worse coverage than *EvoSuite*.
- **Better Time:** Both approaches achieve same coverage, *Evoif* uses less time than *EvoSuite*.
- **Worse Time:** Both approaches achieve same coverage, *Evoif* uses more time than *EvoSuite*.
- **Equal:** This case is none of the case. That is, both approaches achieve the same coverage with the same time.

**4.1.4 Runtime Configuration.** *EvoSuite* provides a rich set of search algorithms, including monotonic genetic algorithm [24, 26], memetic algorithm [26], MOSA algorithm [43], DynaMOSA algorithm [44], etc. In this experiment, we select DynaMOSA algorithm [44] for the following reason.

**Justification.** A successful test evolution towards a target branch requires that the search algorithm addresses multiple challenges



**Figure 5: Overall comparison of *Evoif* vs *EvoSuite* on methods with interprocedural flags.**

at the same time, for example applying appropriate mutations, designing effective optimization algorithm, etc. Therefore, choosing a lesser search algorithm causes that both *Evoif* and *EvoSuite* stuck on the *orthogonal* challenges, which prevents us from comparing *Evoif* and *EvoSuite* effectively. For example, MOSA algorithm will generate test cases for an unnecessary large number of optimization goals, making the Pareto frontier too large to select discriminative test cases to evolve. Thus, we only select the DynaMOSA algorithm which has been evaluated to outperform all other algorithms provided by *EvoSuite* [44]. DynaMOSA is a state-of-the-art multiple objective evolutionary algorithm. It outperforms single objective evolutionary algorithms and traditional multiple objective evolutionary algorithms on various coverage criteria. Its key advantage lies in the capability of dynamically prioritizing some uncovered branches on runtime to improve the testing efficiency.

We run our experiment on 14 nodes on NCL cloud in Singapore (<https://ncl.sg/>), each node is with Intel Xeon E5-2620 CPU of 2.1GHz and 64G DDR4 Memory. The detailed *EvoSuite* runtime configurations can be found in our tool website [1].

## 4.2 RQ1: Prevalence of IPF Problem

We observe that there are 78973 individual methods with program branches in all 150 projects, and 9306 (i.e., 11.8%) of them suffer from interprocedural flag problems. Moreover, we check the ratio of methods suffering from interprocedural flag problem for each project. The average ratio of methods that have at least one interprocedural flag is 14.4% and the median ratio is 11.9%. A more detailed project-wise IPF distribution can be referred in [1]. *In this regard, we conclude that the interprocedural flag problem is prevalent.*

## 4.3 RQ2: Performance

Fig. 5 shows how the comparisons are distributed among the 5 categories. Overall, 255 methods fall to “better coverage”, 57 fall to “better time”, 414 fall to “equal coverage”, 31 fall to “worse time”, and 50 fall to “worse coverage”. We can see that (1) the number of methods in *Better Coverage* is much larger than that in *Worse Coverage* (more specifically, 255 (i.e., 31.6%) v.s. 50 (i.e., 6.2%), (2) the used time is similar for those methods where both *Evoif* and *EvoSuite* achieve same coverage (more specifically, 12.1s v.s. 14.5s on average

```

1  public void solvePhase1(final SimplexTableau tableau) throws
2      OptimizationException {
3          ...
4          if (!MathUtils.equals(tableau.getRhs(), 0, this.epsilon)) {
5              throw new NoFeasibleSolutionException();
6          }
7
8          public static boolean equals(double x, double y, double eps) {
9              return equals(x, y) || FastMath.abs(y - x) <= eps;
10         }

```

**Listing 6: Trivial Interprocedural Flag**

and 4s v.s. 5s on median), and (3) there are a noticeable number of methods where both approaches have equal performance (more specifically, 51.3%). In terms of average coverage, our approach achieves 51.1% coverage comparing to 47.9% of *EvoSuite*. The Mann-Whitney u test on coverage shows that the two-tailed significance value  $p < 0.0001$ , indicating that our improvement on the coverage is statistically significant.

**Qualitative Analysis.** The results show that *Evoif* outperforms *EvoSuite* in general (improves the performance on 312 methods, a.k.a, 38.7% of the total methods) as the algorithm can search towards a valid test case with recovered gradients. Nevertheless, there are still a number of methods where *Evoif* does not improve or even “underperforms” *EvoSuite*. We empirically investigated the details and qualitatively analyzed the worse and equal coverage.

**Equal Coverage and Time.** In this experiment that, despite that 414 (i.e., 51.3%) of the target methods fall into *equal* category, we observe that, in *equal* category, both *Evoif* and *EvoSuite* achieve 100% coverage with less than or equal to 5s on 63 methods, and more importantly, both approaches achieve 0% coverage (and use up the time budget) on 193 methods. The former shows that the interprocedural flag problem is sometimes trivial and can be covered with random guess (see example in Listing 6, as long as the *eps* variable is set to a large value, the condition of interprocedural flag on line 3 is easy to satisfy). The latter case (i.e., 0% coverage) lies in that *EvoSuite* framework has trouble on initializing the constructor of the class declaring the target method. Such an implementation limitation largely impairs the effectiveness of *Evoif*.

**Worse Coverage.** We investigate the target methods where *Evoif* underperforms *EvoSuite*. It happens usually when the interprocedural flag branch  $b_i$  depends on non-interprocedural branch  $b_n$ . Note that, *Evoif* may have a larger runtime overhead than *EvoSuite* as our approach requires more instrumentation to maintain the context information to make sure the branch distance is context-sensitive. Therefore, in general, *Evoif* needs to spend more time to cover “normal” branch before *Evoif* gets a chance to “break through” the tough interprocedural flag branch. Listing 7 shows an example from IDA SDK project [3], we need to break through the branch on line 3 before the approach of *Evoif* takes effect on the branch on line 4. Note that, *EvoSuite* applies rule-based testability transformation on this case so that `String.equals()` method return continuous branch distance and *EvoSuite* can cover it within a number of trials. Nevertheless, such a heuristic testability transformation rule cannot work on the branch on line 4. Noteworthy, during our investigation for this example, *Evoif* will have the same coverage performance with *EvoSuite* with 200s time budget and outperforms *EvoSuite* with a time budget of 250s.

```

1  boolean getMethod(String p1, String p2){
2      ...
3      if (p1.equals(p2)
4          && signatureCorrect(p1)) {
5          return false;
6      }
7  }

```

**Listing 7: Normal Parent Branch**

**Worse Time.** Trivial interprocedural flag branch is the major reason for *Evoif* to achieve its coverage with worse time. Of the 37 target methods where *Evoif* underperforms *EvoSuite* in time, both approaches achieve 100% coverage within 10s on 30 of them. It means that those flags are trivial and can be covered with a few seconds.

#### 4.4 RQ3: Runtime Overhead

During this experiment, within the budget of 100s, *Evoif* evolves on average 498.9 iterations. In contrast, *EvoSuite* evolves on average 680.9 iterations. A more detailed runtime overhead distribution can be referred in [1]. As discussed above, the performance overhead of *Evoif* is caused by building the context-sensitive branch distance. In general, the trade-off between runtime overhead and gradient recovering is paid off as *Evoif* achieves higher coverage even with less number of evolving iterations.

In summary, we conclude that (1) *Evoif* outperforms *EvoSuite* in general (*EvoSuite* achieves higher average coverage and the improvement is statistically significant), (2) *Evoif* may not improve or even underperforms *EvoSuite* when the interprocedural is trivial or some non-interprocedural flag branch guards the interprocedural flag branch, and (3) the gradient recovering technique incurs acceptable overhead (less evolving iterations but higher coverage) for interprocedural flag problem.

#### 4.5 Discussion

**4.5.1 Applicability.** Our experiment empirically shows that testing can be an optimization problem in theory, but code synthesizing problem in practice. The success of applying a search-based software testing in practice involves the careful design of multiple components, for example a well-designed fitness to gauge a “fitter” test case, a comprehensive mutation capability on generated test cases, an efficient optimization algorithm (like DynaMOSA), etc. In this work, we target a finer design of fitness, which presents its effectiveness within the reach of *EvoSuite*’s mutation capability. More specifically, the fitness can only work if *EvoSuite* can provide enough search space to explore. Our approach works well on more target methods with primitive parameters (including String) while is nullified when the target method requires deep analysis on complex data structure and polymorphism. We deem that addressing an orthogonal problem such as improving mutation capability has beyond the scope of this work. Nevertheless, we are targeting to improve mutation capability in our future work.

**4.5.2 No Free Lunch.** From an information perspective, it is not free for us to recover the gradient for branch distance for interprocedural flag problem. Our context-sensitive approach essentially

trades some additional runtime overhead for the gradient information. The experiment shows that such trade-off is paid-off in general. Nevertheless, the price may sometimes be paid in vain, which leads to worse coverage. It is the common problem for all online learning technique used in software testing or fuzzing approach. A more practical and economical way can be a hybrid testing approach combining *Evoif* and *EvoSuite*. Intuitively, we can first apply light-weight *EvoSuite* with a short time budget to cover those trivial branches. Then, for the remaining uncovered “tough” branches, we can then apply *Evoif*. We will explore the optimal way to switch between different testing strategies in our future work.

#### 4.6 Threats to Validity

Threats to external validity arise from our selection of benchmarks. We conducted our experiment on the 150 open source Java projects. Although the number of projects is large, the selected projects may not be representative, and further experiments with more methods are needed to generalize our results. Threats to internal validity arise from how the experiments were carried out. We ran *Evoif* and *EvoSuite* for only 10 times on each method, while the effect of randomness for test-generation may require a larger number of runs to offset. However, to address this threat, we applied both tools in large number of Java methods. In addition, the time budget for each comparison is only 100s, which means that some “equal” comparisons may not be equal if we run both approaches for a longer time. Nevertheless, the same problem may arise under any time budget. In the future, we will run experiments with longer time budget and under more search algorithms for more generalized result. Threats to construct validity come from what measure we chose to evaluate the success of our techniques. We compared the performance of *Evoif* with *EvoSuite* in terms of coverage on individual methods, but did not quantify the effects on entire classes. In the future, we will extend our experiments regarding the above threats to generalize our results.

### 5 RELATED WORK

#### 5.1 Search Based Software Testing

Search based Software Testing (SBST) regards software testing as an optimization problem [28]. Miller et al. [41] pioneered the work for generating test cases with parameters of float types. Following their work, SBST is used for a range of software testing problems, including functional testing [14], regression testing [35], mutation testing [33], as well as test case prioritization [49]. The majority of existing works leverage meta-heuristic search algorithms [12] for generating test cases to cover challenging test goals. Given such a framework, researchers aim to cover various test goals (e.g., branch coverage, path coverage, use-def coverage, etc.) with different search strategies (genetic algorithm, hill-climbing algorithm, etc) [6, 15, 24, 42, 48], fitting test representations [11, 15, 30, 53], and the fitness metrics [11, 30, 45, 53, 58]. Readers are referred to survey papers [32, 40] for more details. Following their work, Aleti et al. [7] investigated the fitness landscape characterisation and showed that the most problematic landscape feature is the presence of many plateaus. The interprocedural flag problem is one of the reasons causing such many-plateaux landscape, where the absence of gradients in the search landscape makes any search-based approach

degenerate to a simple random testing approach. Our work aims at recovering the disappeared gradients, which can facilitate and improve many different search-based software testing approaches.

## 5.2 Testability Transformation

Testability transformation [10, 13, 29, 31, 34, 40] is one of the major techniques for addressing flag problems. Researchers use testability transformation to remove flags by replacing boolean condition with non-boolean condition. Harman *et al.* [31] classified the flag problem into 5 levels based on the complication to make testability transformation. Baresel *et al.* [10] and Binkley *et al.* [13] proposed different flag removal approaches to address the level-5 flag problem, i.e., transforming the flag assigned in the loop. However, these papers only focus on the flag problem within procedures, leaving the interprocedural flag problem unaddressed.

One most relevant work comparing to our approach is Li *et al.*'s method transforming approach [34], which recursively transforms all boolean method calls into ones with non-boolean return types. However, their rule-based approach is limited in practice as the number of necessary transforming rules is huge and keeping the rules practical and consistent is complicated, and it is hard to prove that the rules are semantic-preserving for recursive method transformations. In contrast, our approach does not require complicated transformation rules, as we only require instrumentating the called methods and aggregate the branch distance through program analysis. Our experiment shows that, with acceptable runtime overheads, we can achieve good coverage improvement.

## 6 CONCLUSION AND FUTURE WORK

In this work, we proposed a context-sensitive and recursive approach to address the interprocedural flag problem in search-based software testing. Unlike the traditional testability transformation based approaches, our approach only requires instrumentation to get the branch distances inside the called flag methods, and aggregates those branch distances with regard to specific call contexts and iteration contexts.

In our future work, we will explore hybrid testing strategy combining *Evoif* and *EvoSuite* or even other testing technique such as symbolic execution [52], loop summarization techniques [55–57], and formal analysis [9, 16, 17]. Moreover, we will investigate our guided testing techniques on more platforms [38, 39]. Finally, we will explore the testing criteria beyond the coverage for finding more bugs, and integrate testing with various root cause analysis [18, 19, 27, 36, 37, 50, 51, 54].

## ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments and suggestions. This research has been partially supported by the following grants. The National Research Foundation, Prime Ministers Office, Singapore, under its Corporate Laboratory@University Scheme, National University of Singapore and under its National Cybersecurity R&D Program, Singapore Telecommunications Ltd., the National Research Foundation Singapore under its NSoE Programme (NSOE-TSS2019-03 and NSOE-TSS2019-05), the EPSRC project EP/N023978/2, and the National Science Foundation of China (No. 61632015, U1766215, 61833015).

## REFERENCES

- [1] [n. d.] Anonymous Webiste. <https://sites.google.com/view/evoipf/home>. Accessed: 2019-05-13.
- [2] [n. d.] Apache Math. [https://commons.apache.org/proper/commons-math/download\\_math.cgi](https://commons.apache.org/proper/commons-math/download_math.cgi). Accessed: 2020-01-27.
- [3] [n. d.] IDA SDK. <https://www.hex-rays.com/products/ida/support/download.shtml>. Accessed: 2020-01-27.
- [4] [n. d.] JFreechart. <http://www.jfree.org/jfreechart/download.html>. Accessed: 2020-01-27.
- [5] [n. d.] Weka. <https://sourceforge.net/projects/weka/files/weka-3-8/3.8.0/>. Accessed: 2020-01-27.
- [6] Aldeida Aleti and Lars Grunske. 2015. Test Data Generation with a Kalman Filter-based Adaptive Genetic Algorithm. *J. Syst. Softw.* 103, C (May 2015), 343–352. <https://doi.org/10.1016/j.jss.2014.11.035>
- [7] Aldeida Aleti, I. Moser, and Lars Grunske. 2017. Analysing the Fitness Landscape of Search-based Software Testing Problems. *Automated Software Engg.* 24, 3 (Sept. 2017), 603–621. <https://doi.org/10.1007/s10515-016-0197-7>
- [8] Andrea Arcuri. 2013. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 23, 2 (2013), 119–147.
- [9] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser. 2018. Towards Model Checking Android Applications. *IEEE Transactions on Software Engineering* 44, 6 (2018), 595–612.
- [10] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. 2004. Evolutionary Testing in the Presence of Loop-assigned Flags: A Testability Transformation Approach. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 108–118. <https://doi.org/10.1145/1007512.1007527>
- [11] André Baresel, Harmen Sthermer, and Michael Schmidt. 2002. Fitness Function Design to Improve Evolutionary Structural Testing. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO'02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1329–1336. <http://dl.acm.org/citation.cfm?id=2955491.2955736>
- [12] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. 2009. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. 8, 2 (June 2009), 239–287. <https://doi.org/10.1007/s11047-008-9098-4>
- [13] David W. Binkley, Mark Harman, and Kiran Lakhota. 2011. FlagRemover: A Testability Transformation for Transforming Loop-assigned Flags. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 12 (Aug. 2011), 33 pages. <https://doi.org/10.1145/2000791.2000796>
- [14] Oliver Bühlér and Joachim Wegener. 2008. Evolutionary Functional Testing. *Comput. Oper. Res.* 35, 10 (Oct. 2008), 3144–3160. <https://doi.org/10.1016/j.cor.2007.01.015>
- [15] Jeroen Castelein, Mauricio Aniche, Mozhgan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based Test Data Generation for SQL Queries. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 1220–1230. <https://doi.org/10.1145/3180155.3180202>
- [16] Naipeng Dong, Hugo Jonker, and Jun Pang. 2013. Enforcing Privacy in the Presence of Others: Notions, Formalisations and Relations. In *Computer Security – ESORICS 2013*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.), 499–516.
- [17] Naipeng Dong and Tim Muller. 2018. The Foul Adversary: Formal Models. In *Formal Methods and Software Engineering – 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12–16, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 11232, 37–53.
- [18] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. 2016. ORPLocator: Identifying Read Points of Configuration Options via Static Analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 185–195.
- [19] Z. Dong, A. Andrzejak, and K. Shao. 2015. Practical and accurate pinpointing of configuration errors using static analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 171–180.
- [20] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, 1–12.
- [21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. <https://doi.org/10.1145/2025113.2025179>
- [22] Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 178–188. <http://dl.acm.org/citation.cfm?id=2337223.2337245>
- [23] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite: On the challenges of test case generation in the real world. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 362–369.
- [24] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291.

- [25] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.
- [26] Gordon Fraser, Andrea Arcuri, and Phil McMinn. 2015. A Memetic Algorithm for Whole Test Suite Generation. *J. Syst. Softw.* 103, C (May 2015), 311–327. <https://doi.org/10.1016/j.jss.2014.05.032>
- [27] Wang Haijun, Xie Xiaofei, Li Yi, Wen Cheng, Li Yuekang, Liu Yang, Qin Shengchao, Chen Hongxu, and Sui Yulei. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM.
- [28] M. Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Future of Software Engineering (FOSE '07)*. 342–357. <https://doi.org/10.1109/FOSE.2007.29>
- [29] Mark Harman, André Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. 2008. *Formal Methods and Testing*. Springer-Verlag, Berlin, Heidelberg, Chapter Testability Transformation: Program Transformation to Improve Testability, 320–344. <http://dl.acm.org/citation.cfm?id=1806209.1806220>
- [30] M. Harman and J. Clark. 2004. Metrics are fitness functions too. In *10th International Symposium on Software Metrics, 2004. Proceedings*. 58–69. <https://doi.org/10.1109/METRIC.2004.1357891>
- [31] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Stamer, André Baresel, and Marc Roper. 2004. Testability Transformation. *IEEE Trans. Softw. Eng.* 30, 1 (Jan. 2004), 3–16.
- [32] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2012. *Empirical Software Engineering and Verification*. Springer-Verlag, Berlin, Heidelberg, Chapter Search Based Software Engineering: Techniques, Taxonomy, Tutorial, 1–59. <http://dl.acm.org/citation.cfm?id=2184075.2184076>
- [33] Y. Jia and M. Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. 249–258. <https://doi.org/10.1109/SCAM.2008.36>
- [34] Yanchuan Li and Gordon Fraser. 2011. Bytecode Testability Transformation.
- [35] Z. Li, M. Harman, and R. M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237. <https://doi.org/10.1109/TSE.2007.38>
- [36] Yun Lin, Jun Sun, Lylly Tran, Guangdong Bai, Hajun Wang, and Jinsong Dong. 2018. Break the Dead End of Dynamic Slicing: Localizing Data and Control Omission Bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 509–519.
- [37] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-Based Debugging. In *Proceedings of the 39th International Conference on Software Engineering*. 393–403.
- [38] Tiamming Liu, Haoyu Wang, Li Li, Guangdong Bai, Yao Guo, and Guoai Xu. 2019. DaPanda: Detecting Aggressive Push Notifications in Android Apps. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 66–78.
- [39] Kulani Mahadewa, Kailong Wang, Guangdong Bai, Ling Shi, Jin Song Dong, and Zhenkai Liang. 2019. Scrutinizing Implementations of Smart Home Integrations. *IEEE Transactions on Software Engineering* (2019).
- [40] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.
- [41] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 223–226. <https://doi.org/10.1109/TSE.1976.233818>
- [42] Duy Tai Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Minh Quang Tran. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. 1–12.
- [43] A. Panichella, F. M. Kifetew, and P. Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.
- [44] A. Panichella, F. M. Kifetew, and P. Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.
- [45] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBE '15)*. Springer, 93–108.
- [46] U. Rueda, T. E. J. Vos, and I. S. W. B. Prasetya. 2015. Unit Testing Tool Competition – Round Three. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. 19–24. <https://doi.org/10.1109/SBTS.2015.12>
- [47] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.
- [48] Anupama Surendran and Philip Samuel. 2017. Evolution or Revolution: The Critical Need in Genetic Algorithm Based Testing. *Artif. Intell. Rev.* 48, 3 (Oct. 2017), 349–395. <https://doi.org/10.1007/s10462-016-9504-8>
- [49] Kristen R. Walcott, Mary Lou Sofya, Gregory M. Kapfhammer, and Robert S. Roos. 2006. TimeAware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 1–12. <https://doi.org/10.1145/1146238.1146240>
- [50] Hajun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining Regressions via Alignment Slicing and Mending. *IEEE Transactions on Software Engineering* (2019), 1–1.
- [51] Hajun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating Vulnerabilities in Binaries via Memory Layout Recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 718–728. <https://doi.org/10.1145/3338906.3338966>
- [52] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering*. 291–302.
- [53] Joachim Wegener, André Baresel, and Harmen Stamer. 2001. Evolutionary test environment for automatic structural testing. *Information & Software Technology* 43, 14 (2001), 841–854. <http://dblp.uni-trier.de/db/journals/infosof/infosof43.html#WegenerBS01>
- [54] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology* 105 (2019), 17–29.
- [55] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 61–72.
- [56] Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. 2017. Loopster: static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 84–94.
- [57] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 188–198.
- [58] Xiong Xu, Ziming Zhu, and Li Jiao. 2017. An Adaptive Fitness Function Based on Branch Hardness for Search Based Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, New York, NY, USA, 1335–1342. <https://doi.org/10.1145/3071178.3071184>

# Scalable Build Service System with Smart Scheduling Service

Kaiyuan Wang

Google, USA

kaiyuanw@google.com

Greg Tener

Google, USA

gtener@google.com

Vijay Gullapalli

Google, USA

vijaysagar@google.com

Xin Huang

Google, USA

xnh@google.com

Ahmed Gad

Google, USA

ahmedgad@google.com

Daniel Rall

Google, USA

drl@google.com

## ABSTRACT

Build automation is critical for developers to check if their code compiles, passes all tests and is safe to deploy to the server. Many companies adopt Continuous Integration (CI) services to make sure that the code changes from multiple developers can be safely merged at the head of the project. Internally, CI triggers builds to make sure that the new code change compiles and passes the tests. For any large company which has a monolithic code repository and thousands of developers, it is hard to make sure that all code changes are safe to submit in a timely manner. The reason is that each code change may involve multiple builds, and the company needs to run millions of builds every day to guarantee developers' productivity.

Google is one of those large companies that need a scalable build service to support developers' work. More than 100,000 code changes are submitted to our repository on average each day, including changes from either human users or automated tools. More than 15 million builds are executed on average each day. In this paper, we first describe an overview of our scalable build service architecture. Then, we discuss more details about how we make build scheduling decisions. Finally, we discuss some experience in the scalability of the build service system and the performance of the build scheduling service.

## CCS CONCEPTS

- Software and its engineering;

## KEYWORDS

Build service system, build scheduling service, build system design

## ACM Reference Format:

Kaiyuan Wang, Greg Tener, Vijay Gullapalli, Xin Huang, Ahmed Gad, and Daniel Rall. 2020. Scalable Build Service System with Smart Scheduling Service. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395363.3397371>

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397371>

## 1 INTRODUCTION

Building is a central phase in the software development process in which developers use compilers, linkers, build files and scripts to assemble their code into executable units. Modern build tools like Gradle [6], Buck [2] and Bazel [1] support projects in multiple languages and build outputs for multiple platforms. Programming is often described as an “edit-compile-debug” cycle in which a programmer makes a change, compiles, and tests the resulting binary, then repeats the cycle until the program behaves as expected. Slow builds may cause the programmer to be distracted by other tasks or lose context, and reduce the number of code submissions per day. Any delay increases the gap between the programmer deciding on the next change to perform and viewing the effect of that change.

Although developers can run their builds on their local workstation, the ability to build software remotely is also critical. For example, the build environment of the local workstation may be different, e.g. flags, platforms and build speed, and that difference may cause a build to succeed on one machine but fail on the other machine. A remote build allows us to control the environment and make sure the build result is consistent and reliable. Moreover, many critical services rely on a remote build service. For example, the Continuous Integration (CI) service needs to run builds on remote machines. The software release service also needs to build software every day to make sure products can rollout on time.

To provide a fast and smooth software building experience, we have designed a build service system that is able to run tens of millions of remote builds per day. We list some challenges the build service system needs to solve at Google:

- **Scalability.** The system should be able to handle hundreds of build requests per second and run millions of builds per day.
- **Low Latency.** We don't want to slow down the developers productivity, so the build service overhead should be small and reasonable. The build execution time should be reasonably fast.
- **Reliability.** Since the number of builds is significantly higher than the number of machines to run those builds, we need to reliably keep track of created builds and run them whenever a machine is available. Moreover, since the number of machines is large, machine failure is not a matter of if, but when. So we need to tolerate machine failures and be able to retry the build whenever that happens.
- **Priority.** The builds should be prioritized. Some builds are more important and should be scheduled to run sooner than other builds. For example, a human triggered build is often more important than an automation tool triggered build, because it is more costly for a human to wait.

```

java_library(
    name = "Greeter",
    srcs = ["Greeting.java"],
)
java_library(
    name = "HelloWorld",
    srcs = ["HelloWorld.java"],
)
java_binary(
    name = "HelloWorldMain",
    main_class = "HelloWorld",
    runtime_deps =
        [":HelloWorld"],
)
java_library(
    name = "HelloWorldTest",
    srcs =
        ["HelloWorldTest.java"],
)
java_test(
    name = "AllTests",
    size = "small",
    tags = ["requires-gpu"],
    deps = [":HelloWorldTest"],
)

```

**Figure 1: Example build specification**

- **Instant Feedback.** We want to provide a remote build service that behaves like the build is running on the developer’s local workstations. This means the build output feedback should be streamed back to the user when the build is running. It is unacceptable to deliver the final console output only after the build is completed.

In this paper, we describe the architecture of the build service system used in Google that solves all of the above challenges. Then, we describe more details about our build scheduling service. Finally, we show the scalability of our build service system and the importance of the scheduling service. Specifically, our build service system is able to run 15 million builds on average per day.

The paper makes the following contributions:

- It demonstrates the possibility to implement a build service system that runs tens of millions of builds to support industry-scale development.
- It presents the architecture of the build service system after years of refinement in Google. Our experience could be useful for others to build a scalable build service system.
- It discusses our build scheduling service and its improvement in efficiency on the build service system.

## 2 BACKGROUND

This section describes the Bazel build tool [1], the Spanner database [9] and the proportional–integral–derivative controller [16] we use in our build service system.

### 2.1 Bazel

Bazel is an open-source build and test tool similar to Make [5], Maven [7], and Gradle [6]. It uses a human-readable, high-level build language. Bazel supports projects in multiple languages and builds outputs for multiple platforms. Google uses a build tool built on top of Bazel. For simplicity, we refer to Bazel in the rest of the paper as the build tool in Google.

Bazel takes as input a set of targets that programmers declare in build files. Figure 1 shows a Java sample build specification. Each build specification contains a set of targets. Most targets are one of two principal kinds, files and rules. A rule specifies the relationship between inputs and outputs, and the actions to build the outputs. Actions are sets of system calls, e.g. shell scripts, that will be

```

CREATE TABLE BuildTable (
    id STRING NOT NULL,
    state State BLOB NOT NULL,
    build Build BLOB NOT NULL,
    priority Priority NOT NULL,
    request_time TIMESTAMP
) PRIMARY KEY (id);

CREATE INDEX StatePriorityRequestTime
    ON BuildTable(state, priority, request_time);

```

**Figure 2: Spanner example schema**

executed to complete the build. Rules can be of one of many different kinds or classes, which produce compiled executables and libraries, test executables and other supported outputs. For example, `java_library` is a rule to compile Java source files into libraries. `java_test` is a rule to execute Java tests. `java_binary` is a rule to create executable files for Java. Targets may have dependencies and each target and its dependent targets form an acyclic dependency graph. For example, the Java library target `HelloWorld` depends on the Java library target `Greeter`. The Java library target `HelloWorldTest` depends on `HelloWorld` target. The Java test target `AllTests` depends on `HelloWorldTest` target and the test execution requires GPU. The Java binary target `HelloWorldMain` depends on target `HelloWorld`. When a programmer issues a command to build a target, the build system first ensures that the required dependencies of the target are built. Then, it builds the desired target from its sources and dependencies.

A build includes an execution context, a build command, and some metadata. The execution context specifies the workspace (with or without unsubmitted code changes) in which to run Bazel. We use execution context and workspace interchangeably in the rest of the paper. A Bazel build command specifies the command name, the flags and the target to run. For example, the command `bazel run --jvmopt="-Xms256m" :HelloWorld` runs the `:HelloWorld` target with the JVM startup heap size set to 256 MB. Note that `run` is the command name, `--jvmopt="-Xms256m"` is the build flag, and `:HelloWorld` is the target. The metadata stores other relevant information of the build, e.g. the build’s unique identifier or priority, etc. A majority of our remote builds is one of the three kinds: (1) executing a dependency graph query; (2) build the specified targets; and (3) build and run the specified test targets.

### 2.2 Spanner

Spanner is a scalable, globally-distributed database [9]. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [13] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Figure 2 shows an example Spanner database schema that creates a BuildTable for storing the build to run. The id column stores a universally unique identifier for each build. The state column stores an enum that specifies the running state of the build. The build column stores an object that specifies the build command, flags and targets to run. The priority column stores an enum that specifies the priority of the build. The request\_time column stores the request time of the build. The primary key of the BuildTable is the id column, and it lets Spanner automatically index the BuildTable.

Spanner allows us to create secondary indexes for other columns. Adding a secondary index on a column makes it more efficient to look up data in that column. For example, Figure 2 shows a secondary index StatePriorityRequestTime on the BuildTable, which allows us to quickly find all running builds and iterate over those builds in priority order. For builds of the same priority, we order them by their request time in chronological order. This enables an efficient build scheduling algorithm based on build priority and request time.

### 2.3 Proportional Integral Derivative Controller

A proportional–integral–derivative (PID) controller [16] is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an error value  $e(t)$  as the difference between a desired set point (SP) and a measured process variable (PV), and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively). The input to the process is the output from the PID controller, and it is called manipulated variable (MV).

The proportional, integral, and derivative terms are summed to calculate the output of the PID controller, which is denoted by  $u(t)$  and defined as follows:

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

where  $K_p \geq 0$ ,  $K_i \geq 0$  and  $K_d \geq 0$  denote the coefficients for the proportional, integral, and derivative terms, respectively.  $e(t) = SP - PV(t)$  denotes the error.  $t$  denotes the present time.  $\tau$  is the variable of integration and it ranges from 0 to the present time  $t$ .

In a real implementation, the integral term is discretized, with a sampling time delta  $\Delta t$ , as

$$\int_0^{t_k} e(\tau) d\tau = \sum_{i=1}^k e(t_i) \Delta t$$

and the derivative term is approximated as

$$\frac{de(t_k)}{dt} = \frac{e(t_k) - e(t_{k-1})}{\Delta t}$$

The PID controller is designed to make PV smoothly approach SP and finally equal to SP. In this paper, the build service uses the PID controller to make scheduling decisions.

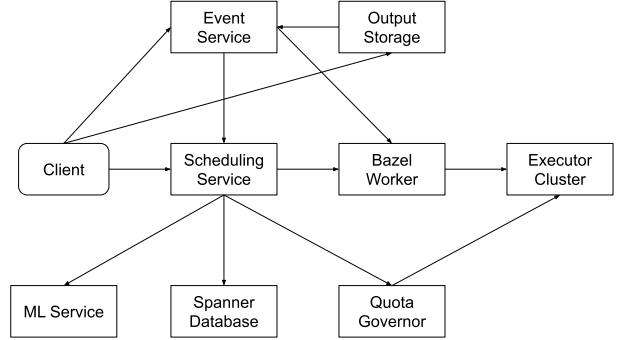


Figure 3: Build service architecture

## 3 BUILD SERVICE ARCHITECTURE

Figure 3 shows the architecture as microservices of the build service system in Google. Each component talks to its connected component via remote procedure calls (RPC). This allows the system to be loosely coupled, scalable, and highly maintainable and testable. Each arrow points from a component to another dependent component. For example, the build scheduling service depends on the Spanner database but not vice versa. In this section, we first briefly describe a common workflow of running a build. Then, we describe each component in the architecture.

### 3.1 Workflow

A typical workflow of a build includes the following steps:

- The client sends the build to the scheduling service.
- The scheduling service sends the build to the Bazel worker and returns a build ID to the client.
- The client uses the returned build ID to query the status of the build via the build event service.
- The worker starts a workspace with a Bazel process. The Bazel process analyzes the build and computes the actual actions to run on the executor cluster.
- The executor cluster runs the actual actions and returns the output.
- The output is sent to the build event service and saved in the storage service.
- The client listens to the build event service as if the build is executing on the client’s local workstation.

### 3.2 Client

The client is provided with a set of APIs to interact with the build service system. Table 1 shows the build APIs the client is able to invoke via RPCs. The CreateBuild API allows the client to create a build in the system. The build will be queued in the scheduling service and executed some time in the future. The API returns a unique build ID so that the client can query the status of the build via the WatchBuild API. The GetBuild API allows the client to query the build and its current state. The CancelBuild API cancels a previously created build, which frees up some system resources. The WatchBuild API allows the client to receive the build output when the build is running. All of GetBuild, CancelBuild and WatchBuild take the build ID as input.

**Table 1: Build API**

API	Description
CreateBuild	Create a build, queuing it to be executed at some point in the future. A unique ID is returned to the client for querying the build.
GetBuild	Get a build, including its current status.
CancelBuild	Cancel a previously created build. If the build is done or in the process of cancelling, this has no effect.
WatchBuild	Return a stream of events for a build.

### 3.3 Scheduling Service

The build scheduling service implements the `CreateBuild`, `GetBuild` and `CancelBuild` APIs. It uses a priority queue to hold all created builds. All builds are ordered by their priority which is inferred from the client that creates builds. For example, builds created by humans for code submission may have higher priority than builds created by an automation tool that collects code coverage. A build is dequeued whenever a Bazel worker is available and there exist enough idle executors to run the build. Higher priority builds are strictly dequeued before lower priority builds. The scheduling service generates a global unique ID for a build, and the build ID is returned to the client.

The scheduling service updates the build state during the lifecycle of a build. Table 2 shows all possible states of a build during its lifecycle. A build can be in one of the three states, i.e. `ENQUEUED`, `IN_PROGRESS` or `FINISHED`. When a build is in the priority queue, it is in the `ENQUEUED` state. When a build is dequeued and running on the Bazel worker, it is in the `IN_PROGRESS` state. When a build is finished and the result is returned, it is in the `FINISHED` state. The scheduling service publishes the `BUILD_ENQUEUED`, `INVOCATION_STARTED` and `INVOCATION_FINISHED` events, respectively, to the build event service whenever the build state is set, for the first time, to `ENQUEUED`, `IN_PROGRESS` and `FINISHED`, respectively. These events which are published when a build state is changed are referred to as *lifecycle* events. Normally, a build moves from the `ENQUEUED` state to the `IN_PROGRESS` state, and finally to the `FINISHED` state. But it is possible that the build is lost during the execution, e.g. the worker dies, so in practice the build can move to the `ENQUEUED` state and then move to the `IN_PROGRESS` state repetitively, and finally move to the `FINISHED` state. Whenever a build is dequeued, the scheduling service generates a unique invocation ID for the execution. The build ID is unique during a build's lifecycle but a build can have multiple invocation IDs. The invocation ID can be used to query build execution result from the output storage service.

### 3.4 Bazel Worker

The Bazel worker is a container with multiple workspaces, and each workspace can run a single Bazel process. Each worker has a limited number of workspaces and thus cannot run more builds in parallel than the workspace capacity. The worker controls the

amount of resources, e.g. memory, network and CPU, assigned to each workspace.

The worker sends a long running `GetNextBuild` RPC to the build scheduling service and receives back the build information to start the invocation. The `GetNextBuild` RPC is sent to the scheduling service whenever a workspace becomes available. The selected workspace then sets up the building environment and starts a Bazel process to run the build. During the execution, the workspace periodically renews its lease on the build, i.e. it notifies the scheduling service that the build is in progress. If the worker dies, e.g. hardware maintenance, and the scheduling service does not receive the lease renewal for a while, then the build will be requeued in the scheduling service. If the workspace tries to renew the lease but the build is not in the `IN_PROGRESS` state, e.g. build is requeued or cancelled, then the worker immediately cancels the build to free up resources. Once the build finishes, the workspace sends the build's final result via the `FinishBuild` RPC to the scheduling service which then sets the build to the `FINISHED` state.

Bazel publishes build events to the build event service during build execution. Some build events contain console output which will be printed in the client's terminal. Other build events contain the verbose build logs and these events will be consumed by the output storage service.

Bazel analyzes the build flags and targets to generate a set of actions. Actions may depend on other actions, e.g. the output of one action may be the input of another action. Thus, Bazel generates a directed acyclic action graph, where a node represents an action and an arrow points the output of one action to the input of another action. Bazel sends the actions to the executor cluster in the topological order in parallel to minimize the execution time. Internally, Bazel caches the output of each action by its input digest and only sends the action to the executor cluster if Bazel does not know the output of that action. The actual executions happen on the executor cluster.

### 3.5 Executor Cluster

The executor cluster executes the actions Bazel sends. The cluster has a global action cache, which can return the action output immediately if the action is executed recently for some other builds. The number of actions running on the executors is very large such that over 99% of actions are cached on average. The executor cluster contains a queue of actions waiting for execution until some executor is free. The action queue length is intended to be small because the scheduling service already has a build queue to hold excessive builds when the executor cluster is full.

The executors have multiple hardware architectures. A majority of the executors are x86 CPUs which are used by all builds. A large set of builds uses Mac machines because Apple's App Store apps must be built on iOS devices. GPUs and TPUs are popular because they can speed up training machine learning models. We use *executor type* to denote different executor architecture in the rest of the paper.

In Google, each product area (PA) is only allowed to use a restricted set of executors so that any PA will not use up all executors. For example, mobile app developers will not see slow builds and be blocked by video website developers who use a large set

**Table 2: Build states and corresponding events**

<b>State</b>	<b>Event</b>	<b>Description</b>
ENQUEUED	BUILD_ENQUEUED	The build is in the priority queue waiting to be dequeued.
IN_PROGRESS	INVOCATION_STARTED	The build is dequeued to a worker and running.
FINISHED	INVOCATION_FINISHED	The build is finished and the build result is returned.

of executors. However, if executors in a PA is underutilized, then another PA can borrow a limited subset of executors from the PA.

Not all actions are equally expensive. For example, some action may require more CPU/memory to run compared to another action. An action that uses one executor with 5GB memory is more expensive than another action that uses one executor with 2GB memory. In the rest of the paper, we use an executor service unit (ESU) to unify the expense of both memory and CPU. One ESU is equal to 2.5GB of memory or 1 executor. The executor cluster keeps track of the type and amount of ESU each build uses and passes that information to another build service component, i.e. quota governor.

### 3.6 Quota Governor

In order to govern the quota usage, each PA is assigned to a limited amount of ESU quota. The intention is to avoid the case where builds from a single PA occupy the entire executor cluster while builds from other PAs cannot be started due to lack of resources. We use PA and quota group interchangeably in the rest of the paper. The quota governor computes the ESU each quota group is allowed to use, and the scheduling service uses that to decide the number of builds to dequeue for each quota group.

Section 2.3 introduces the PID controller. In the quota governor, PV maps to the current executor occupancy, i.e. the amount of ESU the executor cluster is using, per quota group and executor type. SP maps to the desired executor occupancy, i.e. the amount of ESU allowed to use, per quota group and executor type. Each quota group is allowed to borrow unused executor quota from all other quota groups, so the SP of each quota group keeps changing as the utilization ratio of the executor cluster changes. MV maps to the target executor occupancy, i.e. the amount of ESU we want to fill up by running more builds. In practice, the PID controller computes the MV and the scheduling service uses MV to decide how many more builds should be dequeued. When PV is larger than SP and MV is negative, the scheduling service simply stops dequeuing more builds and waits until some resources are freed. When PV is smaller than SP again, the scheduling service starts dequeuing more builds. This strategy makes PV smoothly approaching SP and avoids overshooting.

### 3.7 Build Event Service

The build event service is designed to facilitate build progress reporting. It asynchronously sends build-related events from remote build components (called event publishers) to any number of build watchers. Examples of build events include, but are not limited to, lifecycle events from the scheduling service or console output events from Bazel. A build watcher can watch a build while it's ongoing, or for up to a given period of time after a build has been

created. A build watcher can pause the watch session at an arbitrary position, and then resume the paused session.

Lifecycle events are build level events because they are published when the build state is changed. Bazel events are invocation level events because a build can be lost for many reasons and executed multiple times. No Bazel event can be sent to a build watcher unless both BUILD\_ENQUEUED and INVOCATION\_STARTED events have already been sent. Likewise, the INVOCATION\_FINISHED event cannot be sent to a build watcher unless all its Bazel events have been either completely sent, or considered as expired, i.e. the publisher of this stream probably crashed.

Internally, the build event service uses Spanner to store build events. The scheduling service publishes lifecycle events and Bazel publishes Bazel events. Events are published in the order of event occurrence on the client side. The order of lifecycle and Bazel events are checked on the server side. The build event service allows users to query build events via the WatchBuild API, and it provides both lifecycle and Bazel events in chronological order at real time.

The build event service allows the build watchers to specify event filters which causes only relevant events to be sent to the build watcher. This is particularly useful because the size of all of the events for a build could be huge, e.g. up to hundreds MBs depending on the build, and clients often do not need all events. For example, a command line tool only needs to print the console output to the users. Showing other outputs will likely overwhelm developers. On the other hand, the storage service needs to save all build outputs to facilitate debugging, so it watches all events.

### 3.8 Build Output Storage Service

The build output storage service is a centralized repository for build and test results at the invocation level. It also stores test-related metrics, such as running time and failures. The storage service watches all remote builds, aggregates the build results and displays them to human users. The service keeps the build output for a longer period of time compared to the build event service, which allows users to check the build output of an old build or analyze the build status history pattern. Since the total size of the build output per day is very large, the service does not provide a permanent storage solution. The service provides a UI where users can query their build execution results by providing the invocation IDs.

## 4 BUILD SCHEDULING SERVICE

The clients enqueue builds to the scheduling service and the scheduling service dequeues them whenever some workers and executors are available. The worker keeps pulling builds whenever some workspaces are idle, so the scheduling service only needs to decide if it should send the build to the worker based on the executor availability. The scheduling service dequeues enough builds that fill up the target executor occupancy provided by the quota governor.

In this section, we describe the build scheduling service in more detail. We first describe how to determine some critical properties of the build. Then, we describe why and how the Spanner database is used. Finally, we describe the build enqueueing service, the dequeuing service, and the expiring service.

## 4.1 Critical Build Properties

The build scheduling service conceptually partitions builds into different subgroups based on some build properties. Some build properties are used to sort builds in the priority queue. Those critical properties include build state, quota group, executor types and build priority. The critical build properties do not change during the lifecycle of the build.

The build state is described in Section 3.3 and only ENQUEUED builds are considered for dequeuing.

The quota group is determined by the quota governor, and its value is set to the PA of the actual user that creates the build. The user could be human or an automation tool.

The executor types of a build (before actually running the build) are computed from the build flags (e.g. --ios indicating Mac build), the tags that specifies executor type in any of the build targets (e.g. requires-gpu tag in Figure 1), and the executor type usage history of the build target rules (e.g. since swift\_binary targets use a lot of Mac executors in the past, so any build with swift\_binary target would use Mac executors). Note that all builds use x86 executors but they can use more executor types.

The build priority is derived from the tool used to create the build. Table 3 shows all possible build priorities and some examples. EMERGENCY builds take the highest priority and are dequeued immediately. INTERACTIVE builds are waited by human users and should be dequeued in few a seconds. AUTOMATED builds are important but no human user is waiting at the moment, so these should be dequeued in a couple of seconds to minutes. BATCH builds are not important and can be delayed for a long time. In practice, EMERGENCY builds are rare. Most INTERACTIVE and AUTOMATED builds are often created and dequeued at peak hours (9am-5pm). BATCH builds are dequeued and executed outside of peak hours.

## 4.2 Spanner Database

The build scheduling service uses the Spanner database to keep track of any build state change. The reason to use the Spanner database is to avoid data loss or invalid system states caused by server shutdown. In production, we run multiple scheduling servers to enqueue and dequeue builds, and it's not rare that some servers are shut down by the server manager for maintenance [15]. When the server is back up, it can continue enqueue or dequeue builds without worrying about losing builds or handling invalid build state caused by the last shutdown. Thus, the Spanner database allows us to build stateless servers.

We use a Spanner secondary index (Section 2.2) to create a queue of builds ordered by build state, quota group, build priority and request time. Only ENQUEUED builds are considered for dequeuing. Conceptually, the scheduling service keeps a queue of enqueued builds per quota group, and the queue is ordered by the build priority from highest to lowest. Within the same priority band, the builds are sorted in chronological order of their requested time.

When a build is first enqueued in the database or its state is changed by a transaction, Spanner automatically reindexes the build to the correct position at the transaction commit time.

## 4.3 Build Enqueueing Service

The enqueueing service provides the CreateBuild, GetBuild and CancelBuild APIs. We run many enqueueing service jobs across multiple geographical locations to (1) handle a large number of requests/queries per second (QPS); (2) avoid multiple machine failures in a single machine cluster; and (3) route build requests to the nearest geographical location to reduce network latencies.

The CreateBuild API is the entry point to the build service system. After the client calls CreateBuild with the build to run, the enqueueing service first generates a globally unique build ID using type 4 UUID [14]. Then, the service finds all build properties, e.g. quota group and executor types. Next, the enqueueing service uses a machine learning model to predict the estimated ESU of the build. This helps the dequeuing service to know how expensive the build is and decide if there is enough resource to run the build at the moment. Finally, the enqueueing service persists all build information to the database and marks the build as ENQUEUED, and returns the unique build ID to the client.

Predicting the estimated ESU occupancy of a build is important to determine if the build can be dequeued. If the sum of the estimated occupancy of all in-progress builds is less than the target occupancy given by the quota governor, then the dequeuing service would dequeue some builds to fill up the gap. Otherwise, the dequeuing service would wait until some builds are finished before dequeuing new builds. This strategy ensures that the executor cluster is busy running actions if there are builds waiting in the queue.

We use a linear regression model in TensorFlow [8] to predict the ESU each build uses. Each executor type has a separate model. We train our models using the data in the past 17 days. The data is split into 95% training and 5% testing. The labels are the actual ESU usage of finished builds, which is stored to the disk by the executor cluster. The training pipeline runs continuously and deploys the new models every day. This helps the system to capture the most recent data distribution, and be able to handle builds with new flags or targets. The feature space includes the followings:

- Build command name and flags. Build command name includes build and test, etc. For example, the build command compiles, links and creates code libraries. The test command runs the tests. Build flags include --copt and --android\_sdk, etc. For example, the --copt flag specifies the options passed to the C compiler. The --android\_sdk flag specifies the Android SDK and library used to build Android apps. We treat build flag features as categorical features and each feature can have multiple values. Flags can have out of vocabulary (OOV) values as well. The intuition is that the build flags can affect the actions Bazel generates and the ESU used by the build.
- Build targets and packages. The build target feature is the full path to the target, e.g. path/to/package:target. Each target has a unique path which is the path to the package that contains the build specification followed by the actual target name declared in the spec. The build package feature is simply the target prefix without the target name. Targets and packages are multivalent

**Table 3: Build priorities**

Priority	Description	Example
EMERGENCY	The build is critical and should be executed immediately.	A build that is required for an emergency production bug fix.
INTERACTIVE	The build is important and some human user is waiting for the build.	A presubmit build that blocks code submission.
AUTOMATED	The build is important but no human user is waiting for the build.	A postsubmit build that checks if all tests pass at a given commit.
BATCH	The build is not important and whether it is executed or not blocks nothing.	A nightly build that computes code coverage for some submitted code.

features which can have multiple values for each build. Moreover, we use the target count and package count as numeric features. The intuition is that some targets are significantly more expensive than others, and some packages may contain more expensive targets. Builds with more targets or packages are likely to be more expensive.

- Build properties like quota group and build priority. We found that some PAs may have different ESU occupancy patterns compared to other PAs. For example, the machine learning PA often sends builds that require more GPU/TPU ESUs than other PAs. Build priority is also useful to differentiate build occupancy patterns. BATCH builds typically occupy more ESUs than builds of other priorities, because tools that use BATCH priority, e.g. coverage analysis tools, often send more expensive builds.

We use an off-the-shelf automated blackbox optimization tool similar to AutoML [12] to search for a set of features from the entire feature space that gives the lowest average loss. The selected feature set also includes synthetic feature crosses [4] of up to 6 basic features per cross. For example, one of the feature crosses we use is the combination of command name, iOS sdk version, XCode version and package, which turns out to be a useful feature for predicting Mac ESUs a build may use.

#### 4.4 Build Dequeuing Service

Conceptually, each quota group has a priority queue of builds. The dequeuing service selects the queue using the weighted random selection. Since x86 executors are used a lot more often than other types of executors, each quota group weight is proportional to the x86 ESUs capacity of each quota group. All quota group weights sum up to 1. This means that builds from the PA with more x86 ESU quota are more likely to be attempted for dequeuing.

Once the build queue of a given quota group is selected, the dequeuing service reads a limited number of builds at the head of the queue. The reason to not read the entire queue of builds is that the total number of builds in a queue could be very large and iterating over all builds in the queue would delay dequeuing builds of other quota groups. Note that if we keep dequeuing the builds from the head of the queue, all builds in the queue will eventually be dequeued.

Given a quota group, algorithm 1 illustrates how the dequeuing service decides which builds can be dequeued. The algorithm takes as input a list of builds from the head of the queue *buildsToDequeue*, the target occupancy for each executor type *targetOccupancyByType*, the total estimated occupancy of in-progress builds

**Algorithm 1: Dequeuing algorithm**

**Input:** List of builds *buildsToDequeue*; Target occupancy *targetOccupancyByType*; Total estimated occupancy of in-progress builds *inProgressOccupancyByType*.

**Output:** Dequeueable builds.

```

1 dequeueableBuilds = []
2 reservedOccupancyByType = defaultdict(int) // default value is 0
3 foreach build ∈ buildsToDequeue do
4   isThrottled = False
5   foreach type ∈ getExecutorTypes(build) do
6     targetOccupancy = targetOccupancies[type]
7     inProgressOccupancy = inProgressOccupancyByType[type]
8     reservedOccupancy = reservedOccupancyByType[type]
9     remainingOccupancy = targetOccupancy -
10    inProgressOccupancy - reservedOccupancy
11    buildOccupancy = getEstimatedOccupancy(build, type)
12    if remainingOccupancy < buildOccupancy then
13      isThrottled = True
14    reservedOccupancyByType[type] += buildOccupancy
15  if not isThrottled then
16    dequeueableBuilds.append(build)
17
18 return dequeueableBuilds

```

for each executor type *inProgressOccupancyByType*. The output is the dequeueable builds *dequeueableBuilds*. The algorithm first sets *dequeueableBuilds* to an empty list and sets *reservedOccupancyByType* to an empty map with a default value of 0. *reservedOccupancyByType* keeps the accumulated estimated occupancy reserved so far for each executor type. For each build in *buildsToDequeue*, we get all executor types of the builds. For each executor type, we get the corresponding target occupancy, total in-progress build occupancy, reserved occupancy so far, and compute the remaining occupancy to fill up. Then, we get the estimated occupancy of the build for the executor type. The estimated occupancy comes from the machine learning model in the enqueueing service. If the remaining occupancy is less than the estimated occupancy of the build, then the build is throttled on that executor type. Next, we reserve the estimated occupancy of the build when considering the next build. If the build is not throttled on any executor type, then it is considered as dequeueable. Finally, we return dequeueable builds.

The main idea of Algorithm 1 is that the available ESUs are reserved by the previously iterated builds even if they cannot be dequeued. This avoids the problem when an expensive high priority

build is always throttled and cheaper low priority builds that appear after the high priority build are always dequeued instead.

Once the service finds all dequeueable builds, it needs to choose an appropriate workspace on a Bazel worker, or create one if necessary. A well-chosen workspace can increase the build speed by an order of magnitude by reusing the various cached results from the previous execution. The total number of workspaces of all Bazel workers is very large so it is highly likely that we can find a workspace that previously executed a very similar build, thus reducing the amount of work needed to execute the current build. We have observed that builds that execute the same targets as a previous build are effectively no-ops using this technique.

The dequeuing service has two kinds of workspace selection algorithms as follows:

- The first algorithm computes a hash of the relevant build information and then compares it with the hash of the previously running build in each workspace. The algorithm dequeues the build to the workspace with a matching hash. The build hash is computed from (1) the code repository branch name at which the build is initiated; (2) the build flags and targets, where they can be in different order but still result in the same hash; and (3) the Bazel version required to run the build. It is worth mentioning that the build hash does not depend on the Bazel command name or the base revision of the change. When the service dequeues a build, it sends the build hash to the workspace. The workspace keeps the hash to be able to compare it with the hash of the next build. The intuition is that the target dependency graph of the new build is similar to that of the old build if both builds share the same branch, flags and targets. The reason to include the Bazel version in the hash is to avoid restarting the Bazel process due to version differences.
- The second algorithm computes a set of hashes based on the build's flags and target prefixes. For a build command `bazel build --flag a/b:t1 a/c:t2`, the algorithm computes a set of hashes for each target. For target `a/b:t1`, the hashes are `hash("--flag", "a/b:t1")` and `hash("--flag", "a/b")`. For target `a/c:t2`, the hashes are `hash("--flag", "a/c:t2")` and `hash("--flag", "a/c")`. Then, the algorithm checks if the hashes of each target prefix ever appear in the target prefix hashes in any workspace. If each set of target prefix hashes overlaps with the set of target prefix hashes in the workspace, then the workspace is selected if the Bazel version matches as well. Otherwise, the workspace is not selected. Once a workspace is selected for dequeuing the build, then the dequeuing service unions the sets of hashes of all target prefixes and passes the resulting set of hashes to the workspace. The workspace keeps the set of target prefix hashes for comparison when participating in the next round selection. This method is effective because of the overlap in dependencies of targets that share common prefixes.

The dequeuing service uses the first algorithm to find a workspace, and falls back to the second algorithm if the first algorithm cannot find a match. If the second algorithm cannot find a workspace as well, then the service creates a new workspace to run the build. If the new workspace makes the worker exceed its capacity, then an unused workspace will be shut down.

It is critical that the Bazel process be kept running for as long as possible, because most caches are lost when the process is shut down. The Bazel worker runs a background process which continuously monitors the memory usage on the worker, and shuts down Bazel processes only when the memory usage on the worker exceeds a certain threshold.

After a build is finished, the worker notifies the scheduling service about the final build status, e.g. succeeded, failed, canceled, etc. Then, the scheduling service sets the build state to `FINISHED`, which is the end of the build lifecycle.

## 4.5 Build Expiring Service

Some low priority builds may stay in the queue for a long time. However, if a build is queued for too long, then the result might be obsolete and no longer valid. The build expiring service expires those builds which stay in the queue for more than a given time period, and sets the build state to `FINISHED`.

Some builds are very expensive and may take a long time to run. We do not want to let a build to run forever, so the build expiring service expires the build invocation if it runs for more than a few hours and sets the build state to `FINISHED`. Often these long running builds are problematic, and requires restructuring the build specification or splitting the build into multiple smaller builds.

Section 3.4 mentions that each Bazel worker periodically renews the lease for the build. This makes sure that the scheduling service knows the build is running normally. If for some reason the build lease is not renewed, the build expiring service expires the lease and sets the build state to `ENQUEUED` again. This allows other workers to get and run the build.

## 5 EXPERIENCE

In this section, we first describe the production setup of the build service system. Then, we describe the scalability of the system. Finally, we describe more details about the usefulness of our occupancy models and workspace selection algorithm.

### 5.1 Production Setup

To scale up the build service system, each component is deployed on multiple servers and those servers are widely spread across the world. The load balancer then routes client requests to the nearest group of servers and evenly distributes the traffic to each server.

Google has many datacenters and each data center has many machine clusters. A metro is one or more datacenters that share the same common metropolitan network infrastructure, routing policy, and other associated network resources. Round trip time between any locations within a metro should not exceed few milliseconds. The core components of the build service system are located in a way to minimize the network latency of the build service. Some example configurations are listed below:

- The dequeuing service reads and writes the Spanner database a lot. So the dequeuing servers reside in the same metro as our Spanner database to maximize the dequeuing speed.
- The Bazel process waits until all build events are published to the build event service before finishing the build. So the build event servers reside in the same metro as the Bazel workers to minimize event publishing latency.

**Table 4: Service API QPS**

Service	Avg	Min	Median	Max
CreateBuild	102-253	0-173	89-244	213-833
GetBuild	526-4325	0-529	308-2712	2276-39466
CancelBuild	0-137	0-1	1-3	0-2841
GetNextBuild	94-223	0-166	89-220	177-529
FinishBuild	93-220	0-172	87-220	169-411
WatchBuild	620-3600	0-1757	735-3926	1165-6331

- The Bazel process interacts with the executor cluster a lot by sending actions and receiving results. So the Bazel workers reside in the same metro as the executor cluster to minimize the accumulative network latencies.

The enqueueing server does not need to be in the same metro as the dequeuing server because both services do not directly interact with each other via RPC. Instead, they share the same Spanner database.

The output storage service does not need to reside in the same metro as the build event service, because the storage service does not block the build execution. Moreover, it is acceptable for developers to wait a couple of more seconds after the build is finished to see the build output, thus the network latency is not critical.

## 5.2 Build Service System Scalability

The entire build service system runs more than 15 millions of builds on average and up to 25 millions of builds per day. We have not load tested the entire build service system, but we believe that it can handle millions of more builds. The build service system supports more than 50,000 developers' daily activities in Google.

Table 4 shows the average, minimum, median and maximum of queries per second (QPS), per service API, over the last 24 hour window between 2019/10/01 and 2019/12/31. The average CreateBuild QPS over the last 24 hours ranges from 102 to 253. On 2019/10/20 Sun at around 3:00am, the CreateBuild QPS reaches the minimum of 102. On 2019/11/26 Tue at around 4:30pm, the CreateBuild QPS reaches the maximum of 253. Typically, all service APIs reach the minimum value on weekends outside of peak hours and maximum value on weekdays within peak hours. The average GetBuild QPS ranges from 526 to 4325. GetBuild has a larger QPS compared to CreateBuild because it is often invoked multiple times to query the build status during the build lifecycle. The average CancelBuild QPS ranges from 0 to 137. CancelBuild is not common in practice, but sometimes it is used to cancel problematic builds, e.g. builds that are too big and cause out of memory errors in servers. The average GetNextBuild QPS ranges from 94 to 223. The average FinishBuild QPS ranges from 93 to 220. The QPS of GetNextBuild and FinishBuild are comparable to CreateBuild because each build often corresponds to a single invocation for each of the CreateBuild, GetNextBuild and FinishBuild APIs. The average WatchBuild QPS ranges from 620 to 3600. WatchBuild has a larger QPS compared to CreateBuild because multiple clients can watch the same build. A single client may also watch the same build multiple times because WatchBuild is a long running API and has a higher RPC failure rate. In summary, Table 4 shows that the build service system is able to handle a large amount of throughput.

**Table 5: Service API latency in milliseconds**

Service	Avg	Min	Median	Max
CreateBuild	111-3k	86-146	114-191	124-48k
GetBuild	15-38	4-16	15-26	28-608
CancelBuild	108-432	26-145	91-208	170-41k
GetNextBuild	106k-235k	28k-102k	98k-166k	199k-2064k
FinishBuild	184-689	131-250	177-297	213-71k
WatchBuild	24k-83k	15k-48k	24k-78k	44k-200k

**Table 6: Build count, size and time spent for each priority**

Metric\Priority	EMERG	INTER	AUTO	BATCH
Build Count (%)	0.4	40.1	46.0	13.6
Build Size (KB)	1.5	5.8	22.5	18.4
Target Count	10	22	143	201
Build Queuing	2.4	31.0	282.8	2561.8
Worker Process	4.1	3.3	4.1	3.8
Action Queuing	0.1	0.5	2.3	5.2
Execution	127.6	89.8	133.4	176.1

Table 5 shows the average, minimum, median and maximum of latency in milliseconds, per service API, over the last 24 hour window between 2019/10/01 and 2019/12/31. The average CreateBuild latency ranges from 111ms to 3s. Typically, CreateBuild is designed to have a small latency for better user experience. The average GetBuild latency ranges from 15ms to 38ms. The GetBuild latency is small because it only involves reading the build information from Spanner and returning it to the user. The average CancelBuild latency ranges from 108ms to 432ms. The average GetNextBuild latency ranges from 106s to 235s, which is significantly longer than other API latencies as expected. The average FinishBuild latency ranges from 184ms to 689ms. The latencies of CreateBuild, CancelBuild and FinishBuild are similar, because they involve read-modify-write [11] Spanner transactions. The average WatchBuild latency ranges from 24s to 83s. The WatchBuild API needs to send back a long sequence of events to the client, so its latency is larger.

All build service APIs offer high availability, i.e. the successful rate of the APIs, to the clients. CreateBuild, GetBuild and WatchBuild offer 99.9% availability. CancelBuild, GetNextBuild and FinishBuild offer 99.5% availability.

It is worth to mention that the build output storage service has 30,000 to 60,000 daily active human users who access their build output via web browsers.

Table 6 shows the average build count in percentages, the average size in KB, and average time in seconds spent at each phase. The data is broken down by build priority and collected between 2019/10/01 and 2019/12/31. The table shows that EMERGENCY builds only account for 0.4% of the total builds. INTERACTIVE and AUTOMATED builds are comparable and they account for 40.1% and 46.0% of the builds, respectively. BATCH builds account for 13.6% of the total builds. The build size reflects the number of flags, targets and size of the metadata. Larger build sizes typically indicate more expensive builds (in terms of occupied ESU). However, builds with a small number of expensive targets could be more expensive than builds with a large number of cheap targets. EMERGENCY builds are smaller in size and they are of 1.5KB on average. INTERACTIVE builds

are larger in size and they are of 5.8KB on average. AUTOMATED and BATCH builds are comparable and they are of 22.5KB and 18.4KB on average, respectively. EMERGENCY, INTERACTIVE, AUTOMATED and BATCH priority builds have on average 10, 22, 143 and 201 targets, respectively. EMERGENCY builds are the highest priority builds so they only spend 2.4s on average in the scheduling service. INTERACTIVE builds are the second highest priority builds and they spend 31.0s on average in the scheduling service. AUTOMATED and BATCH builds are lower priority builds and they spend 282.8s and 2561.8s on average in the scheduling service. It takes roughly the same amount of time (3.3s to 4.1s) for the Bazel workers to prepare the environment before starting the build execution across all build priorities. Similar to the build queuing time, the action queuing time on the executor cluster is smaller for higher priority builds. The average action queuing time for EMERGENCY, INTERACTIVE, AUTOMATED and BATCH builds are 0.1s, 0.5s, 2.3s and 5.2s, respectively. The execution time spent on the executor cluster for all builds are comparable and roughly proportional to the build size. The average execution time for EMERGENCY, INTERACTIVE, AUTOMATED and BATCH builds are 127.6s, 89.8s, 133.4s and 176.1s. The average execution time of EMERGENCY builds is larger because they are biased by a small number of long running builds that execute tests hundreds of times. INTERACTIVE builds are often human triggered builds, and they are typically smaller in size and faster to execute because developers often only run a small set of targets affected by their change. AUTOMATED and BATCH builds are often tool triggered builds and those builds are larger in size and slower to execute.

The executor action cache hit rate is often around 99% which speeds up the build execution a lot. Without the action cache, the build service system would not be able to support millions of builds.

### 5.3 Build Scheduling Service Performance

We have discussed some QPS and latency metrics for the scheduling service (Section 5.2). In this section, we mainly discuss the usefulness of the occupancy model (Section 4.3) and the workspace selection algorithm (Section 4.4).

The error of an occupancy model is defined as the difference between the actual ESU and the estimated ESU. The mean square error and the average absolute error of the x86 occupancy model is 312.5 and 5.7, respectively. The mean square error and the average absolute error of the Mac occupancy model is 522.1 and 8.1, respectively. There are more builds that require x86 executors than Mac executors, so the x86 occupancy model has more training data and is more accurate than the Mac occupancy model. Other occupancy models have a lower accuracy compared to the x86 and Mac occupancy models, because only a small fraction of builds require those executor types and those occupancy models have even less training data. Moreover, the executor action cache also makes it hard for the occupancy model to be accurate. Because the same build can use some ESU when there is no cache hit, or 0 ESU when all generated actions hit the cache.

In practice, we find that the occupancy model is useful in making the target executor occupancy more stable and smooth, which helps keeping the executor cluster fully occupied. If we use a default occupancy model which always returns 1 ESU for each executor type per build, the scheduling service would overschedule

builds which causes some expensive low priority builds to run and those builds would prevent high priority builds from executing.

The workspace selection algorithm only affects builds that can reuse Bazel's target dependency or action cache. We collect data from 21 days before and after this feature is enabled, and find that the algorithm is able to reduce the average build execution time from 64.1s to 55.4s, which is a 13.6% improvement. Note that the data is not collected between 2019/10/01 and 2019/12/31, and the data excludes builds that are configured to clean the Bazel cache before running.

## 6 RELATED WORK

CloudBuild [10] is the closest related work to our build service system. It is Microsoft's distributed and caching build service. The main differences are listed below:

- A build in Microsoft contains coarse-grained projects. In Google, a build contains fine-grained targets.
- CloudBuild's distributed cache is similar to the global cache in the executor cluster described in Section 3.5. Microsoft has many codebases and these codebases have fewer shared dependencies. In comparison, Google has a monolithic codebase which results in higher chances in sharing build targets among builds. So CloudBuild's cache hit rate is lower than our executor cache hit rate.
- CloudBuild executes around 20k builds per day and is used by around 4000 developers. In comparison, our build service system executes more than 15 million builds on average per day and is used by more than 50,000 developers.

Another related work is distcc [3], which is a distributed C/C++ compiler publicly available on GitHub. The unit of work is a preprocessed source code file that is sent over the network and compiled remotely. In comparison, our build service system supports many languages and test executions.

Modern build systems like Maven [7], Gradle [6], Buck [2] are not distributed and do not scale to the problem we face in Google. Both Gradle and Buck support local cache which is similar to Bazel in our build service system.

As far as we know, our build service system handles the largest number of builds daily among all build systems developed in other companies. We also introduce the first build scheduling algorithm that uses machine learning models and PID controllers. We believe our work can be beneficial for designing more scalable build service systems.

## 7 CONCLUSION

In this work we presented what we believe is the most scalable build service system in the world. The system has evolved for many years and is refined with many optimizations. We discuss each component in the build service system as well as the build APIs. More specifically, we discuss the build scheduling service and algorithms to dequeue builds. We show that our build service system handles more than 15 million builds on average daily. We also show the usefulness of our occupancy models and workspace selection algorithm in the build scheduling service. We believe that the architecture and algorithms described in this paper are useful and can help in designing new scalable build service systems.

## REFERENCES

- [1] [n.d.]. Bazel build tool. <https://bazel.build/>.
- [2] [n.d.]. Buck build tool. <https://buck.build/>.
- [3] [n.d.]. distcc. <https://github.com/distcc/distcc/>.
- [4] [n.d.]. Feature Cross. <https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture/>.
- [5] [n.d.]. GNU Make. <https://www.gnu.org/software/make/>.
- [6] [n.d.]. Gradle build tool. <https://gradle.org/>.
- [7] [n.d.]. Maven. <https://maven.apache.org/>.
- [8] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.
- [9] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.
- [10] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft’s distributed and caching build service. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 11–20.
- [11] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.
- [12] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2018. *Automated Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at <http://automl.org/book>.
- [13] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
- [14] Paul J Leach, Michael Mealling, and Rich Salz. 2005. A universally unique identifier (uuid) urn namespace. (2005).
- [15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.
- [16] John G Ziegler and Nathaniel B Nichols. 1942. Optimum settings for automatic controllers. *trans. ASME* 64, 11 (1942).

# Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph

Gang Fan

The Hong Kong University of Science  
and Technology  
Hong Kong, China  
gfan@cse.ust.hk

Chengpeng Wang

The Hong Kong University of Science  
and Technology  
Hong Kong, China  
cwangch@cse.ust.hk

Rongxin Wu

Department of Cyber Space Security,  
Xiamen University  
Xiamen, China  
wurongxin@xmu.edu.cn

Xiao Xiao

Sourcebrella Inc.  
Shenzhen, China  
xx@sbbrella.com

Qingkai Shi

The Hong Kong University of Science  
and Technology  
Hong Kong, China  
qshiaa@cse.ust.hk

Charles Zhang

The Hong Kong University of Science  
and Technology  
Hong Kong, China  
charlesz@cse.ust.hk

## ABSTRACT

Modern software projects rely on build systems and build scripts to assemble executable artifacts correctly and efficiently. However, developing build scripts is error-prone. Dependency-related errors in build scripts, mainly including missing dependencies and redundant dependencies, are common in various kinds of software projects. These errors lead to build failures, incorrect build results or poor performance in incremental or parallel builds. To detect such errors, various techniques are proposed and suffer from low efficiency and high false positive problems, due to the deficiency of the underlying dependency graphs. In this work, we design a new dependency graph, the unified dependency graph (UDG), which leverages both static and dynamic information to uniformly encode the declared and actual dependencies between build targets and files. The construction of UDG facilitates the efficient and precise detection of dependency errors via simple graph traversals. We implement the proposed approach as a tool, VERIBUILD, and evaluate it on forty-two well-maintained open-source projects. The experimental results show that, without losing precision, VERIBUILD incurs 58.2 $\times$  less overhead than the state-of-the-art approach. By the time of writing, 398 detected dependency issues have been confirmed by the developers.

## CCS CONCEPTS

- Theory of computation → Program verification; • Software and its engineering → Software defect analysis; Software maintenance tools; • Social and professional topics → Software maintenance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397388>

## KEYWORDS

build maintenance, build tools, dependency verification

### ACM Reference Format:

Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397388>

## 1 INTRODUCTION

In the development of modern software, nearly every large project resorts to build systems (e.g., GNU MAKE, GNU AUTOTOOLS, BAZEL and NINJA) and the corresponding build scripts to automatically transform the source code into the executable software. Build scripts play a critical role in the build process, as they define the rules on compiling source code, resolving the dependencies of build targets, and packaging the binaries. However, developing build scripts is non-trivial and error-prone. Bugs in build scripts are quite common, in both commercial and open-source projects. For example, Hochstein and Jiao [15] found that 11%–47% of test failures are build-related in scientific software. Seo et al. [34] also observed similar build-failure ratios in C++ and Java projects of Google Inc.

Among various kinds of build errors, dependency-related errors are the most common (accounting for 52.68%–64.71%) [34], and can be summarized into two categories: missing dependencies and redundant dependencies. Both are caused by inconsistency between the *declared dependency* (i.e., the dependency declared in the build scripts [17]<sup>1</sup>) and the *actual dependency* (i.e., the dependency needed in the build time [17]<sup>2</sup>). A missing dependency happens when a given actual dependency is not declared, which leads to failures in the parallel build or an incorrect build results in an incremental build [22]. A redundant dependency happens when no actual dependency corresponds to a given declared dependency, leading to a poor performance in the parallel or incremental builds.

<sup>1</sup>A target X has a declared dependency on a target Y, iff rules in the build scripts declare the dependence from X to Y.

<sup>2</sup>A target X is actually dependent on a target Y, iff Y must be present, built and up-to-date in order for X to be built correctly.

Detecting dependency-related errors is tedious and time-consuming [32]. Various techniques have been proposed to tackle different types of dependency-related errors. We classify these techniques into three categories based on how dependency graphs represent the build targets and their dependencies.

The first category constructs the *static dependency graph* via statically analyzing the build scripts and the executed shell commands [1, 36, 37]. Since the static dependency graph is unsound by including only the explicitly declared dependencies, it is infeasible to detect both the missing and redundant dependencies due to the lack of actual dependencies. Therefore, this category of techniques can only detect bad smells in build definitions (e.g., the cyclic dependency issue).

The second category constructs the *dynamic dependence graph* via monitoring and capturing the actual dependency in build processes [22]. Note that, using only the dynamic dependence graph, it is still infeasible to detect the missing and redundant dependencies, due to the lack of declared dependencies for comparison. Therefore, Licker and Rice [22] proposed a build fuzzing technique, MkCHECK, to infer the declared dependencies in the build scripts by triggering incremental builds. Specifically, by updating an input file of a project and triggering the incremental build, only those targets which have direct or transitive declared dependencies on this input file will be rebuilt. Therefore, by updating every input file one at a time to trigger incremental builds, MkCHECK can eventually infer all the declared dependencies in the build scripts. Not surprisingly, this approach is not efficient for a large project with many source files and complex dependencies. For instance, it takes MkCHECK 51.6 hours to complete all the possible incremental builds to infer the declared dependencies for OpenCV, a project with 3,746 source files in total, in our experiment.

The third category [6] leverages both the static and dynamic dependency graphs and cross-references them to discover the inconsistencies, essentially following the definitions of missing dependencies and redundant dependencies. Despite its potential to improve the efficiency, the unsoundness of the conventional static dependency graph is its Achillie's heel. Specifically, the static dependency graph lacks the *implicitly declared dependencies*, which are the dependencies between targets and those output files of its prerequisites that are not specified as targets. Common root causes of this implicitity include the incorrect or the insufficient output files specified in the build scripts (see Section 5.3.3) and temporary files used during a build. To resolve the unsoundness of the static dependency graph, a tool used in the state-of-the-art technique [6], MAKAO [2], tries to extract the implicitly declared dependencies by parsing the command lines and discovering the files with certain extensions (e.g., ".c"). However, this approach cannot fully resolve the unsoundness. Typical examples are those binary files generated during runtime (files without extension) [2]. Moreover, build scripts may include some commands whose syntax would not be known in advance, and thus it would be impossible to devise an omnipotent parser to extract input and output files from commands. This unsoundness issue would lead to a large number of false positives when detecting inconsistencies [6]. Besides, due to the different granularities in the static and dynamic dependency graphs adopted in the existing work [6], it is difficult to remedy

the unsoundness issue of the static information via the dynamic information.

Our idea is also to leverage both static and dynamic dependency information but in a novel way: instead of cross-referencing two graphs, we construct a unified dependency graph (UDG) such that, for each build target, we are able to, via simple graph traversals, both infer the implicitly declared dependencies and compute the actual dependencies happening at runtime. Any difference in the results of these two traversals signifies potential inconsistencies and generates bug reports. Note that, UDG is novel such that both the declared and actual dependencies are fused in an uniform representation. This makes it feasible to infer the implicitly declared dependencies of a target via monitoring file I/O operations of its prerequisite targets during an actual build. As such, we can guarantee the soundness of the declared dependencies. Building UDG is also non-trivial. We propose a novel build instrumentation approach (See Section 3.1.2) to constructing the actual dependencies with respect to each build target, and encode them with the static dependency graph.

Note that, the design of UDG is not restricted to a certain build system. To demonstrate its generality, we have constructed UDGs for different build systems including make-based systems (e.g., GNU MAKE and CMAKE) and NINJA.

To evaluate the effectiveness of our proposed technique, we implemented a tool, VERIBUILD, and applied it to forty-two well maintained open-source projects. The experimental results demonstrated that VERIBUILD has a good performance in efficiency and precision. In the forty-two mature open source projects, VERIBUILD found 2,498 dependency issues with a low overhead. Of all the detected issues, 2,217 are classified as true issues. By the time of writing, 398 of 703 submitted issues had been confirmed by the original developers.

In this paper, we make the following contributions:

- We present the design and the implementation of VERIBUILD, a novel approach to detecting inconsistencies in the build systems. VERIBUILD is more precise and more powerful than the state-of-the-art approaches.
- We propose a new dependency representation, namely the unified dependency graph(UDG), which faithfully encodes the actual and the declared dependencies between input/output files and each target. Based on the UDG, we can find build dependency-related errors by graph traversal on the UDG.
- We select four typical dependency-related errors and implement four checkers in VERIBUILD. The experimental results show that VERIBUILD achieves a high precision with an acceptable time cost. Our approach is more scalable compared with the state-of-the-art techniques.

The paper is organized as follows. We first present some motivating examples in Section II. Section III illustrates our approach. The implementation details and the evaluation are presented in Section IV and Section V respectively. Section VI discusses some related works and Section VII concludes this paper.

```

1 # Makefile
2 check: test-file test-static test-dynamic
3   ./build/test-file
4   ./build/test-static
5   ./build/test-dynamic
6
7 test-file: test-file.c test-dynamic test-static
8   cc test-file.c mpc.c -lm -o build/test-file
9
10 test-static: test-static.c libmpc.a
11   cc test-static.c -lm -lmpc -static -o build/test-static
12
13 test-dynamic: test-dynamic.c libmpc.so
14   cc test-dynamic.c -lm -lmpc -o build/test-dynamic

```

Figure 1: An Example Makefile from MPC

## 2 PRELIMINARIES

### 2.1 Build Systems and Scripts

Modern software projects consist of a large body of source files, libraries and resources. To build or test a software, developers write build scripts (e.g., Makefiles) to instruct a build tool (e.g., GNU MAKE [10], NMAKE [31], BUILD COP [9]) to execute build commands in the order specified in the scripts. Among all the build tools, GNU MAKE is one of the most widely used build systems [30] and has influenced the design of all its successors. However, GNU MAKE is more prone to errors than others, which has led to the invention of many tools (e.g., AUTOTOOLS [8], CMAKE [27]) that help generate Makefiles automatically. Our research and implementation focus on detecting issues in projects managed by GNU MAKE. However, the method we proposed in this paper can also be applied to other build systems. We refer to GNU MAKE as MAKE for simplicity.

Consider a running Makefile build script example shown in Figure 1 simplified from an open-source project MPC<sup>3</sup>. It contains four dependency rules for four targets: *check*, *test-file*, *test-static*, and *test-dynamic*. Each rule specifies the **target** (e.g., *test-file*), the **prerequisites** for building this target (e.g., *test.c*, *test-dynamic* and *test-static*) and the **recipe**, which is a series of shell commands to build this target (e.g., “*cc test.c mpc.c -lm -o build/test-file*”). Based on the rule, the target has the declared dependency on its prerequisite.

Most build tools, including GNU MAKE, BAZEL and NINJA, support advanced features such as the **incremental build** and the **parallel build** for accelerating the build process. In an incremental build, previous build results are reused and only a subset of targets are rebuilt. For example, if *test-file.c* in Figure 1 is modified, only *test-file* and *check* will be rebuilt. Other targets including *test-static*, *test-dynamic* will not be built. In a parallel build, the build system analyzes the build scripts first to decide which targets can be built simultaneously and distributes tasks to different CPU cores to achieve shorter wall-clock build time. For instance, commands for building the targets, *test-static* and *test-dynamic*, can be executed in parallel since they are independent of each other.

### 2.2 Missing and Redundant Dependencies

Build scripts are error-prone. Wrong dependencies can lead to broken incremental and parallel builds, or even to incorrect and unreproducible builds. Two most common dependency-related errors in build scripts are **Missing Dependency** and **Redundant Dependency**.

```

1 #include "ptest.h"
2
3 void suite_core(void);
4 void suite_regex(void);
5 void suite_grammar(void);
6 void suite_combinators(void);
7
8 int main(int argc, char** argv) { ...

```

Figure 2: Content of test-file.c

**2.2.1 Missing Dependency (MD).** An MD is a dependency for building a target that is not specified in the script. MDs are the most common errors in build scripts. They could lead to an incorrect dependency graph being constructed and can corrupt an incremental build and parallel build.

In Figure 1, executing the recipe for *test-file* will read the source file *mpc.c*. However, *mpc.c* is not specified as a dependency of *test-file* in the build script. If *mpc.c* is modified, the build system will not rebuild *test-file* in an incremental build, which leads to incorrect build results. Another common case of MD is related to header files inclusion. Figure 2 shows the content of *test-file.c* where *ptest.h* is included as a header. The compilation of *test-file.c* implicitly depends on the file *ptest.h*. However, this implicit dependency is not specified in the build script.

**2.2.2 Redundant Dependency (RD).** An RD is a redundant prerequisite specified for a target which the build does not depend on. Similar to an MD, an RD can also mislead the construction of the dependence graph used in a build system. For example, the target *test-file* in Figure 1 has two RDs: *test-dynamic* and *test-static*. However, the build of *test-file* does not rely on any outputs of *test-dynamic* or *test-static*. RDs can cause unnecessary targets to be built in an incremental build. If *libmpc.so* is modified, the build system will mistakenly decide *test-file* should be rebuilt, which is clearly unnecessary. RDs could also reduce the parallelism of a parallel build. In this example, a build system should be able to build the three targets *test-file*, *test-static* and *test-dynamic* in parallel since they do not depend on each other. Due to the RDs, the build system will mistakenly decide that the target *test-file* should be built after the target *test-dynamic* and *test-static* are built, even when there are spare CPU cores.

### 2.3 Building Declared and Actual Dependencies

In VERIBUILD, we propose a new type of dependency graph, namely unified dependency graph (UDG), which uniformly encodes the actual and the declared dependencies in terms of build targets and files. Figure 3 shows the UDG of the example build script in the Figure 1.

The subgraph which consists of the static dependency edges and the static spawn edges in Figure 3 is the conventional static dependency graph. Specifically, a static dependency edge represents the dependency between targets, and a static spawn edge represents the case in which a build target is defined as a file. As discussed in Section 1, the implicitly declared dependencies could be missed by statically parsing the build scripts. For example, the target *check* depends on the target *test-static* which could generate the output file *build/test-static*, and thus *check* has implicitly declared dependency on *build/test-static*. Although the shell script in *check* is a typical compiling command and not difficult to parse in this case, it still

<sup>3</sup><https://github.com/orangeduck/mpc>

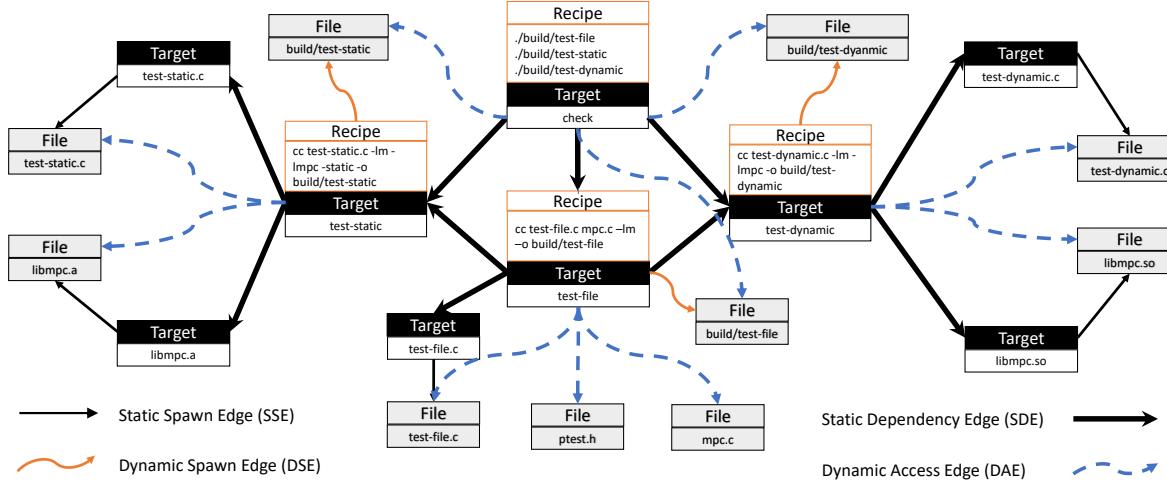


Figure 3: A UDG for the Example Build Script Shown in Figure 1

requires certain engineering effort to customize a parser for this command and is hard to generalize to arbitrary commands. To avoid the parsing trouble, we leverage the build instrumentation (See Section 3.1.2) to capture the output files of each build target (i.e., dynamic spawn edges) in the build time, which can guarantee the soundness of the implicitly declared dependencies. Therefore, a **sound static dependency graph** includes the static dependency edges, the static spawn edges, and the dynamic spawn edges in UDG.

The subgraph which consists of the dynamic access edges and the dynamic spawn edges represents the dynamic dependency graph in terms of build targets and files. Specifically, a dynamic access edge represents an actual dependency between a target and one of its input files, while a dynamic spawn edge represents an actual dependency between a target and one of its output files. Both types of edges are built via our proposed build instrumentation approach (See Section 3.1.2).

## 2.4 Detecting MDs and RDs

With the construction of UDG, we can readily convert the problem of detecting the two common dependency-related build errors, i.e., MDs and RDs, into the graph traversal problem.

Take Figure 3 as an example. To detect the MD issues for the build target `test-file` (missing the declared dependency on `ptest.h` and `mpc.c`), we first traverse from `test-file` along with the edges of the sound static dependency graph to discover the set of files on which `test-file` has declared dependencies, including only `test-file.c`. Then, we traverse along with edges of the dynamic dependency graph to discover the set of files on which `test-file` has actual dependencies, including `test-file.c`, `ptest.h`, and `mpc.c`. From the inconsistency between the two sets of files, we can easily discover that `ptest.h` and `mpc.c` are the missing dependencies for `test-file`.

Detecting the RD issues for the build target `test-file` follows a similar process. First, we discover the actual dependency files for `test-file`, including `test-file.c`, `ptest.h`, and `mpc.c`. Then, to verify whether `test-static` is a redundant declared dependency for `test-file`, we traverse from `test-static` to discover the set of the files on which `test-static` has declared dependencies, including `test-static.c`,

and `libmpc.a`, which do not correspond to the actual dependency files of `test-file`. Based on the definition of the RD issue, we confirm `test-static` is a redundant declared dependency for `test-file`.

The above running examples demonstrate the basic idea of using the UDG to detect the MD and the RD issues. In Section 3, we explain the technical details of the approach, as well as its extensibility to other dependency-related errors.

## 3 APPROACH

We propose a unified approach to analyzing build scripts, which analyzes the static dependency information specified in the build scripts together with the dynamic dependency information to obtain a sound profile of the dependency relations, encoded by a novel data structure, the UDG. The graphical representation of dependency information enables us to analyze build dependencies with graph traversals.

In this section, we first formally define the *UDG* (Section 3.1), followed with a unified construction method. The construction method consists of two steps: *Script Parsing*, parsing and evaluating the build scripts to infer the static dependency graph (Section 3.1.1), and *Build Instrumentation*, instrumenting the build process to infer actual dependencies (Section 3.1.2). Detecting MDs and RDs can be modeled as graph queries on the UDG (Section 3.2). In order to demonstrate the extensibility of our approach, we also discuss two analyses for detecting File-Target Inconsistencies and File Race Conditions (Section 3.3).

### 3.1 Unified Dependency Graph

We first define the Unified Dependency Graph (UDG) as follows:

**Definition 1. Unified Dependency Graph (UDG)**  $UDG = (V, E)$ , where  $V$  is the set of nodes and  $E$  is the set of edges.

$$V = V_T \sqcup V_F: V_T \text{ and } V_F \text{ are two disjoint subsets, where}$$

- $t \in V_T$  is a **target node** that represents a target in the build scripts. Each **target node** is bound with a recipe. A recipe is a list of commands to build the target.
- $f \in V_F$  is a **file node** that represents a file.

$E = E_{SD} \sqcup E_{SS} \sqcup E_{DA} \sqcup E_{DS}$ :  $E$  consists of four disjoint subsets, each of which contains a specific kind of edges defined as follows.

- $e_{SE} \in E_{SE} \subseteq V_T \times V_T$  is a **Static Dependency (SD) edge** from a target node to another (e.g., all the thick solid straight edges in Figure 3), which represents the dependency relation between the two targets specified in the build scripts.
- $e_{SS} \in E_{SS} \subseteq V_T \times V_F$  is a **Static Spawning (SS) edge** from a target node to a file node (e.g., all the thin solid straight edges in Figure 3), which represents the output file for target is statically available before a build.
- $e_{DA} \in E_{DA} \subseteq V_T \times V_F$  is a **Dynamic Access (DA) edge** from a target node to a file node (e.g., all the dashed curved edges in Figure 3), which represents that the file is accessed when building the target.
- $e_{DS} \in E_{DS} \subseteq V_T \times V_F$  is a **Dynamic Spawning (DS) edge** from a target to a file node (e.g., all the solid curved edges in Figure 3), which means that building the target will output the file.

The DA edges represent the actual dependencies, and the SS/DA/DS edges represent the declared dependencies. Provided with a clean build, we can construct the UDG for a project with the following method consists of two phases: **Script Parsing** and **Build Instrumentation**.

**3.1.1 Script Parsing.** We parse and evaluate the build scripts to construct the static dependency graph of a UDG. Parsing user-written build scripts such as Makefiles can be difficult, due to the intensive usages of macros and variables, as well as explicit and implicit pattern rules embedded in build tools. Instead, we parse the build scripts after the build tool has evaluated them (e.g., after macros are expanded and implicit rules are applied). In which, all the rules are specified in the standard form with macros being expanded and evaluated, and all the pattern rules being applied as well. For each of the targets listed in the build scripts, we create one corresponding node. The commands for building a target are kept in the recipe of the target node. For each dependency rule specified in the build scripts, we create SD edges from the target node to all of its dependencies. For each target node with an empty recipe, we check whether its default output file (the file with the same name as the target name) exists in the file system. If the output file exists, we create an SS edge from the target node to a file node representing this file. After script parsing, all the target nodes, SD edges, and SS edges are constructed. File nodes representing the inputs of this build (e.g., source code files and configurations) have been created as well. The resulting graph faithfully describes the static dependencies specified in the build scripts.

**3.1.2 Build Instrumentation.** The purpose of build instrumentation is to obtain the actual input (DA edges) and output files (DS edges) for building a target. There are existing tools based on system call tracing (e.g., *strace* in Linux) and file change monitoring (e.g., *fswatch* in Linux) that can inspect actual file operations for a build. However, the resulting file operations are based on processes (*strace*) or paths (*fswatch*), and cannot easily be correlated to build targets. Thus, instead of monitoring file operations of a build, we perform a build instrumentation, which initiated the build of each target and captures file operations separately.

Based on the static dependency graph obtained in the script parsing phase, we follow the topological order of this graph to build all the targets by executing their recipes. We fork a new process for the execution of each recipe, and capture file-related system calls performed by the forked process and its sub-processes. For each file read operation, we create a DA edge from the target node to the file node. For each file write or create operation, we create a DS edge from the target node to the file node. After build instrumentation, we are able to get the complete UDG.

### 3.2 Dependency Analyses on the UDG

The UDG encodes both the actual dependencies (DA edges) and the declared dependencies (SD/SS/DS edges). It is non-trivial for existing approaches to obtain the files made available when building a target. With the UDG, this problem can be solved by traversing the graph from the target node along with declared dependency edges. All of the reachable files are available at the time of build. This uniform modeling of the declared and the actual dependencies eases the analyses of dependencies. In this subsection, we discuss how to detect MDs and RDs with the UDG. We also discuss two more analyses to demonstrate the broad applicability of the approach (Section 3.3).

**3.2.1 Detection of Missing Dependencies(MDs).** An MD occurs when a target depends on a file that is not declared in the build scripts (transitively). Detecting an MD for a target can be modeled as a graph query on the UDG:

**Problem 1.** For target node  $t$ , if a file node  $f$  is reachable from  $t$  through the DA edges but not through the declared dependency edges (SD edges, SS edges, DS edges),  $f$  is an MD of  $t$ .

To detect MDs for a target, we first analyze its declared dependencies. Algorithm 1 shows two procedures for querying available files for a target: AvFnBefore and AvFnAfter, for querying the files available before and after building a target. In this algorithm,  $t.\text{succs}(\text{edge} = \text{EdgeType}(s))$  denotes querying  $t$ 's direct successors reachable with edges of specific  $\text{EdgeType}(s)$  on the UDG. The resulting file set for each target is cached for optimization purposes.

---

**Algorithm 1** Available file nodes before and after building a target

---

```

procedure AvFnBefore( $t$ : Target Node)
   $P_t := t.\text{succs}(\text{edge} = SD)$ 
   $S_t := t.\text{succs}(\text{edge} = SS)$ 
   $F_t := \bigcup_{p \in P_t} \text{AvFnAfter}(p)$ 
  return  $S_t \cup F_t$ 

procedure AvFnAfter( $t$ : Target Node)
   $S_t := \text{AvFnBefore}(t) \cup t.\text{succs}(\text{edge} = DS)$ 
  return  $S_t$ 
```

---

Algorithm 2 shows how to detect all the MDs for a target, we compare the set of files accessed during its build ( $D_t$ ) to the available file set ( $S_t$ ). All files in  $D_t \setminus S_t$  are MDs to this target.

**3.2.2 Detection of Redundant Dependencies(RDs).** An RD is a declared dependency of a target, which does not occur in the actual build time. Since the build scripts are written in a way where only

**Algorithm 2** Missing Dependency Detection

---

**Input:** UDG, target node  $t$   
**Output:** the MD set of  $t$

- 1:  $S_t = \text{AvFnBefore}(t)$
- 2:  $D_t = t.\text{succs}(\text{edge} = DA)$
- 3:  $MD = D_t \setminus S_t$
- 4: **return**  $MD$

---

one level of dependencies are specified. We only detect RDs in the direct dependencies of a target. The detection of RDs can be modeled as a graph problem on the UDG:

**Problem 2.** If  $t_2$  is a direct dependency of  $t_1$ , for every file node  $f$  that  $t_1$  can reach through both declared dependency edges and actual dependency edges, if we can always find a declared dependency path to  $f$  that does not pass through  $t_2$ , then  $t_2$  is an RD to  $t_1$ .

To decide whether  $p$  is an RD for target  $t$ , we remove  $p$  from  $t$ 's declared dependencies and check whether the declared dependencies in the remaining graph still cover the same actual dependencies when building  $t$ . Algorithm 3 shows the details of the detection. Since the available file nodes of different direct dependencies of a target may overlap, we use a map  $M$  at line 3 to record, for each file node, the number of direct dependencies it is available in. Line 9 to 12 perform the RD detection by checking whether removing a dependency may reduce the number of elements in  $M$ . The underlying problem of RD detection is set-cover [42], which is one of the NP-Complete problems. However, the size of a UDG is relatively small in practice. In the evaluation, the algorithm for detecting RDs always terminates in seconds.

**Algorithm 3** Redundant Dependency Detection

---

**Input:** UDG, Target Node  $t$   
**Output:** the RD set of  $t$ :  $RD$

- 1:  $D_t := t.\text{succs}(\text{edge} = DA)$
- 2:  $P_t := t.\text{succs}(\text{edge} = SD)$
- 3:  $M := \text{Map}(\text{FileNode} \rightarrow \text{Int})$
- 4:  $RD := \emptyset$
- 5: **for**  $p \in P_t$  **do**
- 6:     **for**  $f \in \text{AvFnAfter}(p)$  **do**
- 7:         **if**  $f \in D_t$  **then**
- 8:              $M[f] = M[f] + 1$
- 9:     **for**  $p \in P_t$  **do**
- 10:        **for**  $f \in \text{AvFnAfter}(p)$  **do**
- 11:          **if**  $f \in D_t$  **and**  $M[f] == 1$  **then**
- 12:              $RD = RD \cup \{p\}$
- 13: **return**  $RD$

---

### 3.3 More Analyses on the UDG

With the static and dynamic dependency information encoded in the UDG, we can perform various analyses for other dependency-related errors. We present two analyses for detecting **File-Target Inconsistencies** and **File Race Conditions**, respectively, in the build scripts.

**3.3.1 Detection of File-Target Inconsistencies(FTIs).** In most build systems, a *target* name does not have to be the same as its output file name. For example, many Makefiles have target names such as “all”, “clean” and “test”, however, they do not actually output such files. However, some build systems (e.g., Make) rely on the target name to be the output file name to compute dependencies for incremental build and parallel build. For example, in Figure 1, building target *test-file* will output a file “*build/test-file*” instead of “*test-file*” that is assumed by Make. When building “*test-file*” incrementally, Make will search for file “*test-file*” to decide whether the target *test-file* has been built. Since file “*test-file*” does not actually exist, Make will mistakenly decide that it has to be rebuilt. Moreover, the rebuild of *test-file* will cause all targets that depend on it to be rebuilt, which dooms the purpose of incremental build. We denote such an issue as a File-Target Inconsistency (FTI).

To detect FTIs, we compare the target names with their corresponding output file names. With the UDG, we could easily obtain all the actual output files for a target by traversing along the DS edges. With this method, we can detect that the output file name for target “*test-dynamic*” is “*build/test-dynamic*”. We report it as an FTI issue and suggest the developers to use the filename directly to achieve a better performance.

**3.3.2 Detection of File Race Conditions(FRCs).** Two processes in a parallel build may access the same file for building different targets. If one of the two accesses is a write, a File Race Condition (FRC) might occur. An FRC may cause a parallel build to fail, damage the build artifacts, and results in flaky builds [20].

To detect FRCs for a file, we first identify all the targets that access it and then apply a May-Happen-in-Parallel(MHP) analysis [3, 28] to decide whether those targets may execute in parallel. For a file node, we first check whether it could be accessed by more than one target node. We collect all its predecessors through the DS edges and all predecessors through the DA edges. For any two targets that access this file node, if one is a write access, we check whether there is a path which consists of only SD edges from one to another. If there are no such edges, no order is specified for those two targets in the build scripts, and thus, they may be built in parallel, which could trigger the FRC.

## 4 IMPLEMENTATION

We implement the approach as a tool, VERIBUILD, for analyzing projects maintained with MAKE or NINJA. We also implement the algorithms for detecting four kinds of dependency issues as “checkers” of VERIBUILD. Figure 4 shows the architecture of VERIBUILD. We first infer a dependency graph of all build targets by parsing the build scripts. Second, we instrument a customized builder to build each target according to the static dependency graph in the topological sorting order. When executing the recipe of a target, we trace the system calls to collect the files which are read, written, created, or deleted. After the UDG is constructed, we run the algorithms described in Section 3 to detect dependency issues.

For inferring the static dependency information, we customize a version of MAKE to export its dependency information together with the recipe information, and use the exporting feature supported by NINJA. This exporting feature is commonly supported

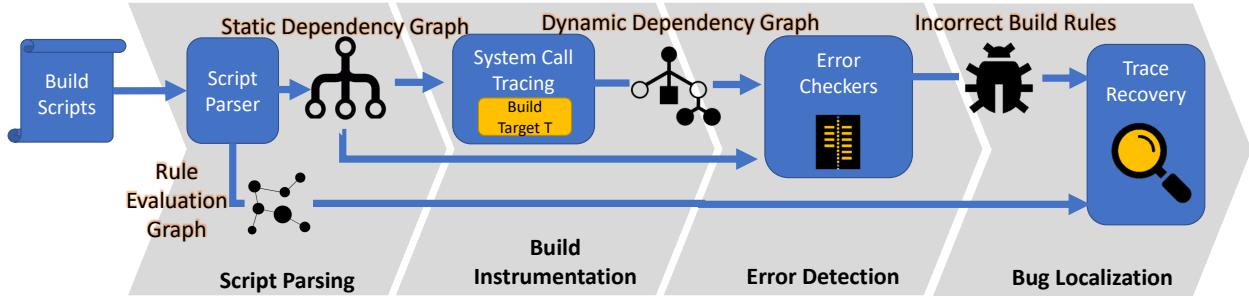


Figure 4: Architecture of VERIBUILD

by many build tools. For example, Make can output its build profile with the option `-profile`, SCONS has a similar option `-tree=all` and BAZEL even provides a query interface for retrieving such build information [19]. Similar to MKFAULT [4], the customized Make also exports the evaluation graph to records the evaluation traces for variables. The evaluation trace is used for assisting developers in localizing the build rules for fixing the detected dependency issues. The discussion of bug localization is beyond the scope of this paper.

To construct the DS/DA edges for a UDG, we need to inspect file operations when executing a recipe. We use `ptrace`, a system call in Unix and Unix-like operating systems that allow one process to inspect and manipulate the internal state of another process. In order to correctly map the dynamic system call traces to corresponding targets, we execute the recipes of targets as described in Section 3.1.2. We fork a new process for executing each recipe and monitor the file-related system calls invoked by the forked process and all of its sub-processes. To be more specific, we capture and inspect system calls such as `read`, `write`, `open`, `unlink`, `execve` and so on.

Using `ptrace` incurs overhead for the processes being traced. Our tool slows down the build by a factor of two. Since we only build the project once for analyzing it, we believe this slowdown is acceptable in practice. An alternative approach would be hooking into the dynamic loading system (e.g., similar to the `ltrace` tool in Linux [21]), which can inspect library calls of dynamic linked libraries and binaries with less overhead. However, this method does not work with the static-linked libraries and binaries.

Note that, our technique has two requirements for build systems. First, the build scripts follow the "target-prerequisite-recipe" model to define the dependencies among build targets. Second, the build systems invoke separate processes for build tasks. Therefore, our technique can be used in the many build systems, including MAKE, NINJA, SCONS, and BAZEL. Our technique could not be applied to Java build systems such as Maven and Ant since they do not satisfy the above requirements. Specifically, Maven and Ant invoke the compiler as a library function call instead of creating a separate process. This makes us unable to extract the actual dependencies by inspecting system calls.

## 5 EVALUATION

### 5.1 Experimental Setup

We test VERIBUILD on forty-two benchmark projects. We compare our approach to MKCHECK [22], since it is the state-of-the-art

technique for incorrect build rule detection. To evaluate the performance of VERIBUILD, we design the following research questions:

- **RQ1:** How effectively does VERIBUILD detect MDs and RDs, compared with MKCHECK?
- **RQ2:** How efficiently does VERIBUILD detect MDs and RDs, compared with MKCHECK?
- **RQ3:** How well does VERIBUILD perform in detecting other build issues that MKCHECK cannot support?

For RQ1 and RQ2, we run both VERIBUILD and MKCHECK on forty-two benchmark projects. For each test, we enable only the checkers for detecting MDs and RDs that both tools support. In order to evaluate the effectiveness and practicability of our approach, we manually inspect and classify the results and also seek confirmation from the developers. In order to evaluate the efficiency of our approach, we compare the overall time cost and the overhead of both tools. For RQ3, we perform the detection of FTIs and FRCs on all the benchmark projects to demonstrate the extensibility of our approach. We manually inspect the reports and perform additional tests to check their validity.

All the experiments are performed on a computer running Ubuntu-16.04 with an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz forty-core processor, and 256GB physical memory.

### 5.2 Subjects

We set up a benchmark set consisting of forty-two projects that are maintained with MAKE (28), CMAKE (12), or NINJA(2), including well-known open-source projects as well as several personal projects evaluated in [22]. Table 1 shows the basic information of these projects. The number of files in those projects ranges from three to over forty thousand. Most of the subjects, such as Python, LLVM, and Redis, are well-maintained in terms of build scripts. For better presentation, we divide the subjects into three categories based on the number of source files: small-sized projects that include less than 100 source files, medium-sized projects that include 100~1,000 files, and large-sized projects that include more than 1,000 files.

### 5.3 Results

**5.3.1 Effectiveness.** To evaluate the effectiveness of our approach, we classify each bug report into two categories: *True Positive* (TP) and *False Positive* (FP). In terms of FPs, we refer to those reports that are unlikely to be fixed by developers. They are due

to developers' intention, dynamic features in the build scripts, or abnormal usages of the build system. To mark a report as a TP or an FP, we employ a three-phased procedure. First, we submit the report to its developers and mark the report as a TP or an FP once the developers confirm or deny it. Second, for an MD report that has not received a response from developers and for the projects with an overwhelming number of reports, we apply an automatic validator to reproduce the reported issue. We mark it as a TP if it is successfully reproduced and an FP otherwise. Third, for RD reports that can not be automatically validated, we manually label it accordingly. As it would be subjective in the last phase, to mitigate this issue, two authors of this paper individually perform checks on the results, and we only consider those reports as TPs if the two authors can reach a consensus. We also present the confirmation results in addition to the TPs and RDs in Table 1 for further inspections.

Moreover, we compare our approach with MkCHECK, the state-of-the-art approach [22]. Note that, MkCHECK and VERIBUILD report MDs and RDs using different dependency relations. MkCHECK describes MD or RD issues by using the dependencies between input/output files, while VERIBUILD describes them by using the dependencies between targets. To ease the presentation, we denote an MD/RD edge as a missing/redundant dependency edge between targets. To ensure a fair comparison, we normalize the reports generated by MkCHECK and VERIBUILD using the same representation using MD/RD edges. For MkCHECK, we try the best to map each file in a bug report into its corresponding target and then collect all the edges between the discovered targets. For VERIBUILD, although we aggregate multiple MD/RD edges starting from the same target into one bug report, we can still recover their original MD/RD edges. In this way, we are able to compare the performance of two tools. Of all the forty-two projects, MkCHECK is able to complete analyzing of 38 projects. Thus, we compare the two approaches with these projects only.

Table 1 shows the results: VERIBUILD detects 2,051 MDs and 437 RDs in forty-two subjects, of which 2,217 reports are classified as TPs. VERIBUILD manages to detect issues in most of those subjects (39/42). To further evaluate its effectiveness, we seek confirmation from the original developers of those projects. However, due to the heavy workload of reporting and communicating, we only submit reports for those projects with less than one hundred reports (except for Python, developers of which are responsive and help us confirm more than one hundred reports). Moreover, to avoid flooding developers with too many bug reports, we manually pre-screen the bug reports and only submit the severe ones. In total, we submit 703 reports to their developers. By the time of writing, 398 of them are confirmed by the developers, and the rest are unconfirmed.

Compare to MkCHECK, VERIBUILD surpasses it in terms of the number of the MD edges and the RD edges in 25 and 22 projects, respectively. The reason why VERIBUILD can detect more RD/MD issues in these projects is that MkCHECK only fuzzes files with some certain extensions and would inevitably miss some true cases for those files with unexpected extensions. We observed that MkCHECK reports more MD edges in eight projects, and more RD edges in two projects, compared with VERIBUILD. We investigated these reports and found that most of them are redundant. For example, if an MD edge  $T_1 \rightarrow T_2$  ( $T_1$  depends on  $T_2$ ) is discovered by MkCHECK,

**Table 1: Classification of Detected Dependency Issues**

Project	# MDs TP/ALL	# RDs TP/ALL	# Total TP/ALL	# Confirm AC/SUMB	# MD Edges VB/MC/=	# RD Edges VB/MC/=
LLVM	29/89	0/67	29/156	0/0	n/a	n/a
OpenCV	40/68	0/16	40/84	0/40	n/a	n/a
Python	153/158	146/153	299/311	138/138	4419/706/679	5406/5399/5079
PHP	383/383	0/34	383/417	0/0	22694/1/0	78/1/1
GMP	512/519	1/1	513/520	0/0	631/329/0	1/0/0
OpenSSL	346/347	0/1	346/348	0/0	10865/0/0	222/0/0
Capstone	47/47	51/58	98/105	98/98	198/5659/138	621/523/523
httpd	1/1	0/0	1/1	0/1	1/2/0	0/122/0
Redis	4/11	4/4	8/15	0/4	160/0/0	9/0/0
Bash	63/65	34/51	97/116	48/48	513/1482/278	378/142/45
Cppcheck	39/41	13/13	52/54	36/36	53/0/0	42/18/18
ck	12/12	1/1	13/13	0/13	181/180/180	3/0/0
tig	0/0	0/0	0/0	0/0	0/0/0	2/0/0
lighttpd	0/1	0/0	0/1	0/0	104/69785/0	0/0/0
Tmux	2/2	0/0	2/2	0/2	2/0/0	0/0/0
neven	85/85	0/0	85/85	0/85	1453/1459/1453	0/0/0
GNU Aspell	0/0	1/1	1/1	1/1	0/12/0	1/0/0
LAME	0/0	1/1	1/1	0/1	0/3/0	2/0/0
cJSON	1/1	0/0	1/1	0/1	159/0/0	58/0/0
zlib	5/6	0/0	5/6	0/5	96/2/2	0/0/0
lee	2/2	0/0	2/2	0/2	2/0/0	0/0/0
clib	44/44	0/3	44/47	44/44	178/147/147	6/6/6
8cc	16/16	1/1	17/17	0/17	16/0/0	1/0/0
fastText	11/11	4/4	15/15	0/15	11/0/0	4/0/0
gravity	23/23	0/0	23/23	0/23	262/0/0	0/0/0
x86-thing	17/17	0/0	17/17	0/17	76/76/76	0/0/0
grbl	16/16	0/0	16/16	0/16	20/20/20	0/0/0
Bftpd	13/14	12/14	25/28	25/25	14/0/0	111/97/97
gwion-util	1/1	0/0	1/1	1/1	18/0/0	10/0/0
cctz	0/0	0/0	0/0	0/0	0/0/0	0/0/0
fzy	6/6	0/0	6/6	0/6	17/17/17	0/0/0
libeo	20/20	0/0	20/20	0/20	32/63/23	0/0/0
mpe	13/13	12/13	25/26	0/12	18/0/0	14/0/0
namespaced_parse	3/3	0/0	3/3	0/3	10/4/3	7/0/0
kleaver	7/7	0/0	7/7	0/7	11/2/0	0/0/0
agt-proof-of-stake	4/4	0/0	4/4	0/4	n/a	n/a
greatest	7/8	0/0	7/8	7/7	8/1/1	1/0/0
Stack-RNN	2/2	0/0	2/2	0/2	14/0/0	0/0/0
CacheSimulator	2/2	0/0	2/2	0/2	3/2/2	0/0/0
conceptnet5.5-paper	1/1	0/0	1/1	0/1	n/a	n/a
http-parser	4/4	1/1	5/5	0/5	4/0/0	1/0/0
Generic-C-Project	1/1	0/0	1/1	0/1	4/1/1	2/0/0
Total	1935/2051	282/437	2217/2488	398/703		

TP = True Positive AC = Accepted SUMB=Submitted n/a = not available VB = Veribuild MC=MkCheck

all of the targets which depend on  $T_1$  will be reported as missing dependency on  $T_2$  correspondingly, which lead to a significant larger number of redundant reports. In summary, other than the implementation issues, VERIBUILD and MkCHECK achieve similar performance, in terms of the effectiveness. To further understand the effectiveness of VERIBUILD, we investigate some true and false MD/RD cases.

**True MDs.** Many MDs are cases of missing source files. They have various causes such as unsynchronized build scripts and source files, misuses of the implicit rules in Makefile, and clerical errors. Figure 6 shows an MD report we generated for *agt-proof-of-stake*<sup>4</sup>, which uses NINJA to maintain its build. Two prerequisites of target *report.pdf* are missing: *cite.bib* and *llncs2e/llncs.cls*. The incremental build of *report.pdf* will be incorrect if any one of these two files is changed. Many other MD reports are the cases in which C/C++ header files are missing. Specifically, in order to decide the right files that a compilation may access, the headers directly and indirectly included in the source file should be carefully considered for the declared dependencies. Figure 5 shows an MD we reported in *fastText*<sup>5</sup>, a library for text classification and representation learning from Facebook. Line 47 in Makefile specifies the dependencies of *model.o*. However, it misses the header file *utils.h* included in *model.cc* and all the headers included in *model.h*. One way to prevent these problems is to use the Auto-Dependency Generation [25]

<sup>4</sup><https://github.com/lucaspena/agt-proof-of-stake>

<sup>5</sup><https://github.com/facebookresearch/fastText>

```

46 # Makefile
47 model.o: src/model.cc src/model.h src/args.h
48 $(CXX) $(CXXFLAGS) -c src/model.cc
49

8  /* model.cc */          8  /* model.h */
9  #include "model.h"       9  #pragma once
10 #include <loss.h>         10
11 #include <utils.h>        11 #include <memory>
12 #include <assert.h>        12 #include <random>
13 #include <algorithm>       13 #include <utility>
14 #include <stdexcept>       14 #include <vector>
15 #include <stdexcept>       15
16 #include "matrix.h"        16
17 #include "real.h"          17
18 #include <utils.h>         18 #include "vector.h"
19 Model::State::State(...)  19
20 : lossValue_(0.0),        20
21 nexamples_(0),           21
22 namespace fasttext {      22

```

Figure 5: An Example MD from *fastText*

```

28 # build.ninja
29 build stage-1.out : maude stage-1.maude
30
31 build report.pdf : pandoc report.md | template.latex
32 flags = --template template.latex --pdf-engine=xelatex $-
33 --filter pandoc-citeproc --bibliography=cite.bib
34

```

Figure 6: An Example MD from *agt-proof-of-stake*

mechanism provided by some compilers, and another way is to use VERIBUILD to periodically check the build scripts.

**False MDs.** False MDs are mainly caused by non-deterministic features in Make. For example, Make supports the use of conditional branches in a recipe. The execution of a recipe may take different branches across different builds, and all are considered valid. However, our approach assumes that every file access leads to unconditional dependency, and thus reports false positive in such cases. Figure 7 shows such an example extracted from *Python*. It specifies all headers (*PYTHON\_HEADERS*) as the dependencies of all objects (*LIBRARY\_OBJS* and *MODOBJJS*), which results in hundreds of RDs being reported (Table 1). Similar to the MDs, the header problems could also be solved with Auto-Dependency Generation or a tool such as VERIBUILD. Other RDs are just the simple mistakes made by developers, such as the RD in Figure 1.

**True RDs.** In order to fix missing dependencies, developers may specify too many extra dependencies. Since correctly handling the C/C++ header dependencies is difficult, many projects specify all headers as the dependencies for all objects. Figure 7 shows such an example extracted from *Python*. It specifies all headers (*PYTHON\_HEADERS*) as the dependencies of all objects (*LIBRARY\_OBJS* and *MODOBJJS*), which results in hundreds of RDs being reported (Table 1). Similar to the MDs, the header problems could also be solved with Auto-Dependency Generation or a tool such as VERIBUILD. Other RDs are just the simple mistakes made by developers, such as the RD in Figure 1.

**False RDs.** When a dependency is identified as an RD of a target by VERIBUILD, it means the build of that target indeed does access any file that exclusively provided by this dependency. However, some of them are unlikely to be fixed by developers due to various reasons. For these RDs, we mark them as False RDs in Table 1. For example, Figure 8 shows the content of *Makefile.in* in *Bash*. VERIBUILD reports *Makefile* and *config.h* are two RDs of the target *bashbug*, since they are not accessed by any process

```

431 # Makefile.pre.in
432 LIBRARY_OBJS= \
433 $(LIBRARY_OBJS OMIT_FROZEN) \
434 Python/frozen.o

962 PYTHON_HEADERS= \
963 $(srcdir)/Include/Python.h \
964 $(srcdir)/Include/abstract.h \
965 $(srcdir)/Include/asdl.h \
966 $(srcdir)/Include/ast.h \
967 ...
968

1095 $(LIBRARY_OBJS) $(MODOBJJS) Programs/python.o: $(
1096   PYTHON_HEADERS)

```

Figure 7: An Example RD from *Python*

```

594 # Makefile.in
595 bashbug: $(SDIR)/bashbug.sh config.h Makefile $(VERSPROG)
596 @sed -e "$!PATCHLEVEL!$%PatchLevel%" \
597 $(SDIR)/bashbug.sh > $@
598 @chmod a+r bashbug
599

945 # Makefile.in
946 test tests check: force $(Program) $(TESTS_SUPPORT)
947 @-test -d tests || mkdir tests
948 @cp $(TESTS_SUPPORT) tests
949 @(` cd $(srcdir)/tests && \
950 PATH=$(BUILD_DIR)/tests:$PATH THIS_SH=$(THIS_SH) $(SHELL) \
951 {TESTSCRIPT} `)

```

Figure 8: An Example RD from *Bash*

when building the target *bashbug*. However, they should not be removed from the rule because it is the developers' intention to rebuild *bashbug* if these two files are modified. The file *config.h* is generated automatically when the project is re-configured, and its change is a flag to tell the whole project to rebuild.

Another category of false RDs are order-only prerequisites. Developers specify the build rules to enforce the build order of different targets, instead of the input-output dependencies. For example, in Figure 8, at line 946 in *Makefile.in*, the developer specifies *force* as the dependency of *test*, *tests* and *check*. The build of the three targets does not rely on any output from building *force*, but the developer intends to build them only after the build of *force*. To be more efficient, one can specify such dependencies as order-only prerequisites [26] supported by Make, and VERIBUILD can utilize this information to filter out false RDs caused by order-only prerequisites.

**Answer to RQ1** VERIBUILD manages to detect 2,051 MDs and 437 RDs in forty-two real open-source projects, of which 2,217 are true bugs reports and 398 reports have been confirmed by the developers, which demonstrates its effectiveness.

**5.3.2 Efficiency.** To evaluate the efficiency, we compare VERIBUILD with MKCHECK, in terms of the time cost. Although the prior study [22] does not report the time cost of MKCHECK, we directly run their tool on all the subjects in our experiment and record the execution time. MKCHECK requires that a target name being specified for constructing the dependency graph. Thus, we run both tools for only the default targets in the build scripts.

Table 2: Performance Data for VERIBUILD and MkCHECK

Origin	Project	# Lines	# Files	Clean Build (sec)	MkCheck(sec)				VeriBuild(sec)				Speedup		
					Dep Graph	Build Fuzz	Race Detection	Overhead	Total	Script Parsing	Build INSTR	MD/RD Detection	Overhead	Total	
(#Files < 100)	Generic-C-Project	15	3	0.11	0.25	6.47	6.37	118.0X	13.09	0.12	0.46	0.03	4.5X	0.61	26.1X 21.6X
	http-parser	6,347	6	33.09	33.84	138.86	167.77	9.3X	340.47	0.04	35.10	0.03	0.1X	35.17	147.5X 9.7X
	conceptnet-5.5-paper	3,041	6	1.26	n/a	n/a	n/a	n/a	n/a	0.12	1.68	0.03	0.5X	1.83	n/a n/a
	CacheSimulator	953	10	0.32	0.71	15.61	2.68	58.4X	19.00	0.04	0.92	0.02	2.1X	0.98	28.3X 19.4X
	Stack-RNN	1,746	10	15.53	8.18	191.22	15.68	12.8X	215.08	0.02	16.02	0.06	0.0X	16.11	346.5X 13.4X
	greatest	1,581	11	0.61	1.21	16.70	11.09	46.5X	29.00	0.03	1.59	0.03	1.7X	1.66	27.2X 17.5X
	agt-proof-of-stake	3,687	12	14.81	n/a	n/a	n/a	n/a	n/a	0.15	15.70	0.05	0.1X	15.90	n/a n/a
	kleaver	734	14	0.72	1.72	35.86	0.97	52.5X	38.55	0.21	2.38	0.04	2.6X	2.63	19.8X 14.7X
	namespaced_parser	599	15	0.62	0.53	8.89	14.96	38.3X	24.38	0.04	0.69	0.02	0.2X	0.75	188.0X 32.7X
	mpc	4,566	16	21.22	24.83	444.39	91.91	25.4X	561.13	0.03	26.79	0.07	0.3X	26.89	95.2X 20.9X
Open Source Projects	fzy	3,798	30	0.81	1.85	28.96	16.41	57.3X	47.22	0.03	1.87	0.04	1.4X	1.95	40.8X 24.3X
	libeo	3,727	31	3.44	7.06	52.49	34.05	26.2X	93.60	0.05	8.05	0.15	1.4X	8.25	18.7X 11.3X
	cetz	7,782	33	14.83	21.46	184.58	64.81	17.3X	270.85	0.10	23.66	0.24	0.6X	24.00	27.9X 11.3X
	gwion-util	1,238	34	0.67	1.65	64.68	19.69	127.4X	86.02	0.09	2.10	0.07	2.4X	2.26	53.7X 38.1X
	Bftpd	5,003	37	1.91	3.17	90.20	26.42	61.7X	119.79	0.03	4.06	0.09	1.2X	4.18	51.9X 28.7X
	grbl	4,514	51	1.21	2.63	108.09	38.11	122.0X	148.83	0.64	3.38	0.09	2.4X	4.11	50.9X 36.2X
	x86-thing	2,903	55	0.08	0.24	99.77	56.03	1,949.5X	156.04	0.08	0.11	0.02	1.7X	0.21	1,171.4X 732.1X
	gravity	17,817	56	9.60	13.27	275.97	58.82	35.3X	348.06	0.13	15.48	0.17	0.6X	15.77	54.8X 22.1X
	fastText	6,749	68	18.74	23.16	249.43	70.15	17.3X	342.74	0.19	26.18	0.21	0.4X	26.59	41.3X 12.9X
	Sec	10,165	82	1.21	2.38	126.71	31.91	132.1X	161.00	0.04	3.32	0.25	2.0X	3.61	66.6X 44.6X
(100 < # Files < 1000)	clib	14,843	105	2.24	5.38	204.99	102.77	138.8X	313.14	0.08	7.27	0.19	2.4X	7.54	58.7X 41.5X
	lee	4,397	106	14.41	24.66	192.59	19.52	15.4X	236.77	0.47	28.67	0.03	1.0X	29.17	15.1X 8.1X
	jlib	28,998	106	7.27	10.23	110.52	84.31	27.2X	205.06	0.07	12.03	0.19	0.7X	12.29	39.4X 16.7X
	cJSON	19,437	107	0.51	0.38	11.89	8.75	40.2X	21.02	0.08	1.15	0.03	1.5X	1.26	27.3X 16.7X
	LAME	47,124	146	8.60	18.70	429.36	224.72	77.2X	672.78	0.81	28.23	0.44	2.4X	29.48	31.8X 22.8X
	GNU Aspell	30,821	184	45.04	66.72	2,760.43	532.27	73.6X	3,359.42	0.97	80.96	1.13	0.8X	83.05	87.2X 40.5X
	neven	22,743	186	7.92	10.79	377.87	182.07	71.1X	570.73	0.15	12.66	0.44	0.7X	13.24	105.7X 43.1X
	Tmux	52,244	206	24.71	44.71	678.33	333.32	41.8X	1,056.36	0.44	63.38	1.70	1.7X	65.52	25.3X 16.1X
	lighttpd	61,072	232	33.09	61.90	1,440.96	483.14	59.0X	1,986.00	3.12	83.29	2.13	1.7X	88.55	35.2X 22.4X
	tig	43,394	238	8.42	16.34	752.33	103.89	102.6X	872.56	0.18	20.46	0.54	1.5X	21.17	67.8X 41.2X
Open Source Projects	ck	31,743	244	1.53	2.84	480.65	28.60	333.7X	512.09	0.40	3.36	0.03	1.5X	3.79	225.4X 134.9X
	Cppcheck	182,121	399	148.61	182.06	3,614.48	437.18	27.5X	4,233.72	0.09	209.75	1.14	0.4X	210.98	65.5X 20.1X
	Bash	126,820	442	41.53	63.57	2,100.20	485.56	62.8X	2,649.33	0.48	89.56	1.32	1.2X	91.36	52.3X 29.0X
	Redis	111,435	457	20.10	27.92	1,440.26	152.70	79.6X	1,620.88	0.43	40.12	0.09	1.0X	40.64	77.9X 39.9X
	httpd	199,327	502	113.54	243.03	1,388.86	1,562.61	27.1X	3,194.50	2.26	398.03	0.36	2.5X	400.66	10.7X 8.0X
	Capstone	161,930	604	36.92	51.51	1,734.99	296.26	55.4X	2,082.76	0.44	71.06	0.49	1.0X	72.00	58.3X 28.9X
	OpenSSL	580,653	1,767	290.03	n/a	n/a	n/a	n/a	n/a	3.68	340.06	18.81	0.3X	362.54	n/a n/a
	GMP	258,875	1,878	87.35	179.13	4,057.00	6,159.08	118.0X	10,395.21	2.72	220.76	10.82	1.7X	234.30	70.1X 44.4X
	PHP	766,944	2,077	521.51	764.33	6,293.62	6,165.85	15.6X	8,673.80	5.78	812.63	7.42	0.6X	825.82	26.8X 10.5X
	Python	1,053,963	2,645	161.87	197.48	29,432.70	1,077.04	188.7X	30,707.22	0.88	264.71	3.26	0.7X	268.84	285.5X 114.2X
(# Files > 1000)	OpenCV	1,670,991	3,746	2,694.86	3,140.96	185,822.75	4,193.23	70.7X	193,156.94	4.13	3744.65	45.54	0.4X	3,794.32	173.2X 50.9X
	LLVM	5,670,236	41,788	19,113.58	n/a	n/a	n/a	n/a	n/a	9.77	21,576.22	435.45	0.2X	22,021.44	n/a n/a

n/a = data unavailable due to the failure of MkCheck.

Table 2 shows the evaluation results. Overall, VERIBUILD is one or two orders of magnitude faster than MkCHECK for almost all subjects. Especially for the large subjects, VERIBUILD achieves speedups from 10×(PHP) to 114×(Python). The time cost of MkCHECK in checking large projects such as OpenCV (193,157s) is high, which hinders its wide adoption. Note that, VERIBUILD achieves such a high performance but does not sacrifice the effectiveness (see Section 5.3.1).

To understand the performance bottleneck of each tool, we further investigate the time cost of each step for both tools. MkCHECK includes three steps, i.e., Dependency Graph Construction, Build Fuzz, and Race Detection. VERIBUILD includes three steps, i.e., Script Parsing, Build Instrumentation, and MD/RD Detection. The results show that both tools incur some overhead for inspecting the system calls for a build (MkCHECK-build v.s. build, VERIBUILD-build v.s. build) and the overheads are in the same level (MkCHECK 59%, VERIBUILD 108%), since both tools use the *ptrace* system call to monitor files operations of a clean build. VERIBUILD incurs slightly higher overhead since it also parses and analyzes the build scripts in this stage.

The testing time of MkCHECK is dominated by the build fuzzing time (91.1%). As we have discussed in Section 2, the primary purpose of build fuzzing is to obtain static dependency information. As a comparison, VERIBUILD parses and evaluates build scripts to obtain the static dependency information, and it costs only 39.37 seconds in total, which is 0.016% (39.37/245,463.66) of the build fuzzing time of MkCHECK. The analysis time cost is relatively trivial for VERIBUILD.

For analyzing all projects, VERIBUILD spends only 533.16 seconds in total for analyzing both MDs and RDs.

**Answer to RQ2** In this experiment, VERIBUILD is 26.5X faster than MkCHECK, on average. Moreover, VERIBUILD only incurs <90% runtime overhead compared with the native clean build (i.e., without instrumentation) on average.

**5.3.3 Extensibility.** To demonstrate that our approach has better extensibility than the fuzzing testing for detecting build-related issues, we have implemented two checkers to check FTIs and FRCs. With the UDG being constructed, the effort for implementing the two checkers are rather trivial. Both checkers are written with less than 60 lines of Python code and have taken less than one hour for one of our authors for the implementation, which demonstrates great extensibility of applying our approach to new build-related problems. We run the two checkers on the forty-two benchmarks we used in RQ1 and RQ2. In total, we detected six FTIs in two projects and eight FRCs in one project. The detailed results can be found on the project's website. Here we only discuss two representative cases.

**File-Target Inconsistency(FTI).** Figure 1 shows a FTI we detect in MPC. The target names “*test-file*”, “*test-static*” and “*test-dynamic*” are inconsistent with the files these targets actually output. Due to these three FTIs, the build system will mistakenly rebuild these three targets even for an incremental build. After correcting these FTIs in the Makefile, the build time for the incremental build is reduced from 21 seconds to 0.03 seconds.

---

```

41 # Makefile
42 PROGS = colib example_poll example_ecosvr example_ecochli
43 ...
44 all:$(PROGS)
45
46 libcolib.a: $(COLIB_OBJS)
47 $(ARSTATICLIB)
48 libcolib.so: $(COLIB_OBJS)
49 $(BUILDSHARELIB)
50
51 example_ecosvr:example_ecosvr.o
52 gcc -o example_ecosvr example_ecosvr.o -g -L./lib -lcolib
53 -lpthread
54 example_ecochli:example_ecochli.o
55 gcc -o example_ecochli example_ecochli.o -g -L./lib -lcolib
56 -lpthread

```

---

**Figure 9: An Example FRC from libco**

**File Race Condition(FRC).** Figure 9 shows an FRC example in *libco*, including the report and the corresponding Makefile. For this case, a file race can occur for building target “all”: the sub-targets *libcolib.a* and *example\_closure* could have data race on file “*libcolib.a*”. We expand the macros in build commands (Line 52 and 54) for better illustration. When building *example\_closure*, the compiler *gcc* will access and read *libcolib.a* (option *-lcolib* instructs the compiler to link *libcolib.a* with *example\_ecosvr.o*). *libcolib.a* is created when building *libcolib.a* and there is clearly no dependency specified for these two targets. We test this FRC by building *libco* with different settings: the build fails for 4/100 times with four concurrent jobs and fails 100/100 times with more than five concurrent jobs.

**Answer to RQ3** We are able to apply the VERIBUILD approach to two practical dependency-related build errors that existing approaches do not support, with less than two hundred lines of code, which, to some extent, demonstrates the good extensibility of VERIBUILD.

## 6 RELATED WORK

**Bug detection in build scripts.** Gunter [12] used a Petri net to check the correctness and detect possible optimization opportunities in Makefiles. Tamrawi et al. [37] proposed SYMAKE which uses symbolic dependency graphs to automatically detect bad smells, such as cyclic dependencies and duplicate prerequisites, in the build scripts. Xia et al. [43] and Zhou et al. [44] adopted data mining techniques to predict missing dependencies. All the above approaches only analyze the information statically available in the build scripts. Therefore, they are unable to detect many dependency issues due to the lack of the actual dependency information.

Another line of research studies is to use dynamic techniques to verify dependencies. Licker and Rice [22] proposed a fuzzing technique to detect dependency issues. It constructs a file-level dependency graph by tracing system calls of a clean build and tests the correctness of build scripts by triggering an incremental build for each file. As discussed in Section 2, build fuzzing is impractical for medium and large projects due to its overwhelming number of incremental builds. Several modern build systems such as FABRICATE [38] and MEMOIZE [29] can automatically avoid missing dependencies by monitoring the building process at runtime. Such a monitoring-based approach inevitably slows down the building

process. BAZEL [18] is a build system that creates an isolated environment for each build task, so that it can avoid writing unexpected results to the file system. ELECTRICACCELERATOR [7] automatically corrects the order of build steps at runtime, rather than detecting RDs and MDs for developers to diagnose. TUP [35], IBM CLEARCASE [16] and VESTA [41] verify dependency issues at runtime. These techniques require that the dependency issues should be triggered first, otherwise, they are unable to detect them. Different from that, our approach can detect dependency issues even before their occurrences.

Using actual dependency information together with the static dependency information in the build scripts has been proven to be effective. Bezemer et al. [6] proposed a tool to detect so-called unspecified dependencies in Make-related build systems. However, unspecified dependencies are not necessarily missing dependencies. Their approach reports 1.2 million unspecified dependencies in four projects, most of which are merely implicit dependencies intentionally omitted by the developers. Our approach also analyzes both the static and the dynamic dependencies but is different from theirs in three aspects: first, our approach performs a more fine-grained analysis that analyzes build targets rather than files; second, the UDG used in our approach encodes implicitly declared dependencies, which are missing from their approach; third, the missing and the redundant dependencies detected by our approach could be directly derived from the unspecified dependencies found by their approach.

**Build error localization.** There are also many research studies proposed for localizing build errors [4, 5, 24, 33, 40] and automatically fixing them [14, 23]. We believe that their techniques are complementary to ours and have the potential for greater effectiveness if combined with VERIBUILD.

**Build maintenance.** A massive research effort has been devoted to improving the maintainability of build systems [2, 11, 13, 37, 39]. FORMIGA [13] supports simple renaming and removal of the build targets. SYMAKE [37] focuses on the renaming and extraction of the build targets. MAKAO [2] uses an aspect-oriented approach to support adding new commands and dependencies. Vakilian et al. [39] developed tools for decomposing build specifications. CLOUDMAKE [11] from Microsoft leverages system call tracing for migrating build systems.

## 7 CONCLUSION

In this work, we have discussed the limitations of existing approaches to incorrect build rule detection. With the insight that existing approaches do not make good use of available static and dynamic dependency information, we have presented a unified approach, which has been demonstrated to be faster, more precise and have better bug-detection capability than the state-of-the-art. We believe that VERIBUILD is promising in industry settings.

## ACKNOWLEDGMENTS

Rongxin Wu is the corresponding author. We thank anonymous reviewers for their constructive comments. This work is funded by Hong Kong GRF16230716, GRF16206517, ITS/215/16FP, ITS/440/18FP grants, and the National Natural Science Foundation of China (Grant No. 61902329).

## REFERENCES

- [1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. 2008. The evolution of the Linux build system. *Electronic Communications of the EASST* (2008). <https://doi.org/10.14279/tu.eceasst.8.115.119>
- [2] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. 2007. Design recovery and maintenance of build systems. In *IEEE International Conference on Software Maintenance (ICSM)*. <https://doi.org/10.1109/ICSM.2007.4362624>
- [3] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOP)*. <https://doi.org/10.1145/1229428.1229471>
- [4] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for make-based build crashes. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME)*. <https://doi.org/10.1109/ICSME.2014.87>
- [5] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault localization for build code errors in makefiles. In *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings*. <https://doi.org/10.1145/2591062.2591135>
- [6] Cor Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering* (2017). <https://doi.org/10.1007/s10664-017-9510-8>
- [7] Electric Cloud. 2020. ElectricAccelerator EMake: Speeds Up Builds and Tests. <https://electric-cloud.com/plugins/directory/p/emeake/> [Online; accessed 25-Aug-2019].
- [8] Automake Contributors. 2012. GNU Autotools. <https://www.gnu.org/software/automake> [Online; accessed 25-Aug-2019].
- [9] Drake. 2020. Build Cop. <https://drake.mit.edu/buildcop.html> [Online; accessed 25-May-2020].
- [10] Stuart I. Feldman. 1979. Make – a program for maintaining computer programs. *Software: Practice and Experience* (1979). <https://doi.org/10.1002/spe.4380090402>
- [11] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdy, and Benjamin Livshits. 2014. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices* (2014). <https://doi.org/10.1145/2714064.2660239>
- [12] Carl A. Gunter. 1996. Abstracting dependencies between software configuration items. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. <https://doi.org/10.1145/250707.239129>
- [13] Ryan Hardt and Ethan V. Munson. 2013. Ant build maintenance with formiga. In *2013 1st International Workshop on Release Engineering, RELENG 2013 - Proceedings*. <https://doi.org/10.1109/RELENG.2013.6607690>
- [14] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE/ACM. <https://doi.org/10.1145/3180155.3180181>
- [15] Lorin Hochstein and Yang Jiao. 2011. The cost of the build tax in scientific software. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*.
- [16] International Business Machines Corporation (IBM). 2020. IBM Rational Clearcase. <https://www.ibm.com/us-en/marketplace/rational-clearcase> [Online; accessed 25-Aug-2019].
- [17] Google Inc. 2019. Actual and declared dependencies. [https://docs.bazel.build/versions/master/build-ref.html#actual\\_and\\_declared\\_dependencies](https://docs.bazel.build/versions/master/build-ref.html#actual_and_declared_dependencies) [Online; accessed 25-Aug-2019].
- [18] Google Inc. 2019. Bazel - a fast, scalable, multi-language and extensible build system. <https://bazel.build/> [Online; accessed 25-May-2020].
- [19] Google Inc. 2019. Bazel Query. <https://docs.bazel.build/versions/master/query-how-to.html> [Online; accessed 25-Aug-2019].
- [20] Lim James. 2019. Combating Flaky Builds. <https://medium.com/@jimjh/combating-flaky-builds-f8aaa9ccd29a> [Online; accessed 25-Aug-2019].
- [21] Petri Machata, Juan Cespedes. 2019. ltrace. <http://man7.org/linux/man-pages/man1/ltrace.1.html> [Online; accessed 25-May-2020].
- [22] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 1234–1244. <https://doi.org/10.1109/ICSE.2019.00125>
- [23] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: How far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. <https://doi.org/10.1145/3293882.3330578>
- [24] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. <https://doi.org/10.1109/SANER.2018.8330201>
- [25] GNU Make Manual. 2019. Auto-Dependency Generation. [ftp://ftp.gnu.org/old-gnu/Manuals/make-3.77/html\\_node/make\\_43.html](ftp://ftp.gnu.org/old-gnu/Manuals/make-3.77/html_node/make_43.html) [Online; accessed 25-Aug-2019].
- [26] GNU Make Manual. 2019. Prerequisite Types. [https://www.gnu.org/software/make/manual/html\\_node/Prerequisite-Types.html](https://www.gnu.org/software/make/manual/html_node/Prerequisite-Types.html) [Online; accessed 25-Aug-2019].
- [27] Ken Martin and Bill Hoffman. 2010. *Mastering CMake: a cross-platform build system*. Kitware.
- [28] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. *Event London* (1989). <https://doi.org/10.1.1.47.7435>
- [29] Bill McCloskey. 2019. memoize. <https://github.com/kgaughan/memoize.py> [Online; accessed 25-May-2020].
- [30] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. 2014. Mining co-change information to understand when build changes are necessary. In *Proceedings - 30th International Conference on Software Maintenance and Evolution (ICSME)*. <https://doi.org/10.1109/ICSME.2014.46>
- [31] Microsoft. 2020. NMAKE Reference. <https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=vs-2019> [Online; accessed 25-May-2020].
- [32] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.
- [33] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden). Association for Computing Machinery, New York, NY, USA, 71–81. <https://doi.org/10.1145/3180155.3180224>
- [34] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at google). In *Proceedings - International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1145/2568225.2568255>
- [35] Mike Shal. 2009. Build system rules and algorithms. *Published online* (2009). Retrieved July 18 (2009), 2013. [http://gittup.org/tup/build\\_system\\_rules\\_and\\_algorithms.pdf](http://gittup.org/tup/build_system_rules_and_algorithms.pdf)
- [36] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland). IEEE Press, 650–660.
- [37] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. 2012. SYMAke: a build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 366–369.
- [38] Brush Technology. 2019. fabricate. <https://github.com/brushtechnology/fabricate> [Online; accessed 25-May-2020].
- [39] Mohsen Vakilian, Raluca Saucic, J. David Morgenhaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE)* (Florence, Italy). IEEE Press, 123–133.
- [40] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C. Gall. 2018. Un-break My Build: Assisting Developers with Build Repair Hints. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)* (2018), 41–4110.
- [41] VestaSys. 2020. Vesta Configuration Management System. <http://www.vestasy.org/> [Online; accessed 25-May-2020].
- [42] Wikipedia contributors. 2020. List of NP-complete problems. [https://en.wikipedia.org/w/index.php?title=List\\_of\\_NP-complete\\_problems&oldid=957698266](https://en.wikipedia.org/w/index.php?title=List_of_NP-complete_problems&oldid=957698266) [Online; accessed 25-May-2020].
- [43] Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. 2014. Build system analysis with link prediction. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. <https://doi.org/10.1145/2554850.2555134>
- [44] Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. 2014. Build predictor: More accurate missed dependency prediction in build configuration files. In *Proceedings - International Computer Software and Applications Conference (COMPSAC)*. <https://doi.org/10.1109/COMPSAC.2014.12>



# How Far We Have Come: Testing Decompilation Correctness of C Decompilers

Zhibo Liu

The Hong Kong University of Science and Technology  
Hong Kong, China  
zliudc@connect.ust.hk

## ABSTRACT

A C decompiler converts an executable (the output from a C compiler) into source code. The recovered C source code, once recompiled, will produce an executable with the same functionality as the original executable. With over twenty years of development, C decompilers have been widely used in production to support reverse engineering applications, including legacy software migration, security retrofitting, software comprehension, and to act as the first step in launching adversarial software exploitations. As the *paramount component* and the *trust base* in numerous cybersecurity tasks, C decompilers have enabled the analysis of malware, ransomware, and promoted cybersecurity professionals' understanding of vulnerabilities in real-world systems.

In contrast to this flourishing market, our observation is that in academia, outputs of C decompilers (i.e., recovered C source code) are still *not* extensively used. Instead, the intermediate representations are often more desired for usage when developing applications such as binary security retrofitting. We acknowledge that such conservative approaches in academia are a result of widespread and pessimistic views on the decompilation correctness. However, in conventional software engineering and security research, how much of a problem is, for instance, reusing a piece of simple legacy code by taking the output of modern C decompilers?

In this work, we test decompilation correctness to present an up-to-date understanding regarding modern C decompilers. We detected a total of 1,423 inputs that can trigger decompilation errors from four popular decompilers, and with extensive manual effort, we identified 13 bugs in two open-source decompilers. Our findings show that the overly pessimistic view of decompilation correctness leads researchers to underestimate the potential of modern decompilers; the state-of-the-art decompilers certainly care about the functional correctness, and they are making promising progress. However, some tasks that have been studied for years in academia, such as type inference and optimization, still impede C decompilers from generating quality outputs more than is reflected in the literature. These issues rarely receive enough attention and can lead to great confusion that misleads users.

\*Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397370>

Shuai Wang\*

The Hong Kong University of Science and Technology  
Hong Kong, China  
shuaiw@cse.ust.hk

## CCS CONCEPTS

- Software and its engineering → Software reverse engineering; Software testing and debugging.

## KEYWORDS

Software Testing, Reverse Engineering, Decompiler

### ACM Reference Format:

Zhibo Liu and Shuai Wang. 2020. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397370>

## 1 INTRODUCTION

A software decompiler recovers program source code by examining and translating executable files. C decompilers are among the most fundamental reverse engineering tools for software re-engineering missions [18, 35, 63], and they have also laid a solid foundation for many cybersecurity applications, including malware analysis and off-the-shelf software security hardening [22, 25, 26]. To date, many C decompilers exist on the market, including commercial tools that cost several thousands of US dollars and also free versions actively maintained by the open-source community. Remarkable commercial decompilers on the market, including IDA-Pro [34] and JEB3 [52], are “must-have” gadgets for reverse engineers and security analysts despite their high prices. Free decompilers maintained by the open-source community, such as RetDec [38] and Radare2 [2], have also started to challenge the dominance of their commercial competitors. Recently, the National Security Agency (NSA) has also released its decompiler framework, Ghidra [48, 50], with the aim of “training the next generation of cybersecurity defenders.”

Despite being the core component of most reverse engineering tasks in industry, our observation is that C decompilers, particularly their final-stage outputs (i.e., the recovered C code), are *not* extensively used in academia. While C decompilers are frequently employed in academia as the basis of analyzing legacy software, such as malware clustering, firmware analysis, and security retrofitting [18, 22, 25, 26, 35, 63, 67], the recovered intermediate information is usually preferred in conducting research rather than decompiled C code. For instance, to fix a security flaw in an executable, the convention is to perform binary patching and to edit the executable file after reverse engineering program layouts and locating the issue [27], even though the straightforward way is to decompile the executable, instrument the decompiled code, and recompile the hardened code into a new executable file.

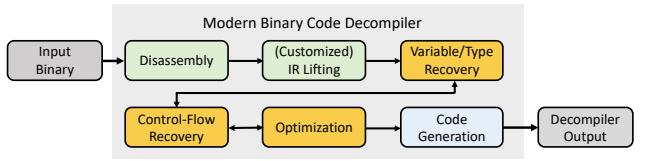
We interpret such *conservative* approaches as being due to the widespread and potentially pessimistic stance on decompiled C

code; one might expect that the decompiled C code is primarily designed as presentable high-level descriptions of input executables, instead of being directly used as a conventional C program. However, recent several years of progressive development on “functionality-preserving” disassembling and C style control structure recovery [17, 31, 47, 64, 65, 67] illustrates that functionality-preserving decompilation of unobfuscated and not-highly-optimized executable is primarily a matter of engineering effort (although a certain amount of readability is sacrificed). Indeed, some popular reverse engineering frameworks are tentatively implementing such techniques to guarantee decompilation correctness in the first place [28, 55]. Overall, we argue that the research community lacks to incorporate an *up-to-date* understanding of de facto C decompilers and the correctness of their outputs, which may impede reaching the full potential of modern C decompilers in conducting research. Thus, this work aims to study C decompilers in a realistic setting and to more clearly delineate the decompilation correctness of modern C decompilers.

In this research, we perform systematic testing to reflect the decompilation correctness of C decompilers, which is certainly not fully understood and can lead to a controversial view between industry hackers and academic researchers. In particular, we target x86 to C decompilers whose inputs are x86 executable files, with such decompilation being deemed as highly challenging and popular. We aim to answer the following important research questions: **RQ1**: *how difficult is it to recompile the outputs of modern C decompilers?*; **RQ2**: *what are the characteristics of typical decompilation defects?*; and **RQ3**: *what insights can we deduce from analyzing the decompilation defects?* Our findings can be adopted to promote the development of decompilers and to serve as guidelines for users to avoid potential pitfalls.

This work is the first to conduct a comprehensive study targeting C decompilers. We employ Equivalent Modulo Inputs (EMI) testing, a random testing technique that has achieved major success in revealing compiler bugs, in this new setting [19, 40, 60]. We also organize a group of security analysts to carry out extensive manual inspection on findings yielded by EMI testing (about 530 man-hours in total). From a total of 10,707 programs used for this study, we found 1,423 programs exposing decompilation errors from four widely-used decompilers, two of which (IDA-Pro [34] and JEB3 [52]) are popular commercial tools, and the other two (RetDec [38] and Radare2/Ghidra [50]) are actively developed and maintained by the community and by NSA. Manual inspection on the open-source decompilers (RetDec and Radare2/Ghidra) detects 13 buggy code fragments that incur the decompilation errors we found. In sum, this research makes the following contributions:

- To study C decompilers in a realistic setting and to delineate their up-to-date capabilities, we introduce and advocate a new focus, conducting comprehensive and large-scale testing on C decompilers. Findings obtained in this study will guide future research that aims to use and improve decompilers.
- We reuse well-established compiler testing techniques in this new setting and form a productive workflow to reveal potential decompiler bugs. From two de facto commercial decompilers and two popular free decompilers, we successfully found 1,423 programs causing decompilation errors.



**Figure 1: The workflow of C decompilers.** We use different colors to differentiate decompilation stages. The key focus is the middle stage (modules in orange) where variables, types, and high-level control structures are recovered.

We have reported all the findings to the decompiler developers, and by the time of writing, typical defects have been promptly confirmed by commercial decompiler vendors and to be fixed. We manually confirmed 13 bugs which caused all errors found in outputs of free decompilers.

- We made various observations and obtained inspiring findings regarding modern decompilers. We show that the overly pessimistic stance on the recompilability leads researchers to *underestimate* the potential of C decompilers; modern decompilers are making encouraging progress to enhance quality of their outputs. In contrast, subtler issues that lead to erroneous and unreadable decompiled outputs, including the type inference failure and over optimization, frequently exist and do not receive enough attention.
- We have released all the findings and our tool for decompiler testing to facilitate further research [8, 9]. Our artifact has passed the ISSTA Artifact Evaluation check and been awarded the Functional badge. Other decompilers can be tested with our tool following the same procedure.

## 2 BACKGROUND OF RESEARCH

### 2.1 Pipeline of C Decompilers

Fig. 1 depicts a high-level overview of modern C decompilers. Multiple stages are involved, and the output of one stage is the input of the next stage.

*Front End: Disassembling.* Input executables will first be fed into the disassembling module to translate binary code into assembly instructions. The data sections within each executable will also be identified for further usage. Existing research of disassembling has been shown to work very well in practice and the proposed methods can smoothly disassemble large-size binary executables [13, 39, 65]. Nevertheless, the disassembled outputs (i.e., assembly code) are seldom used for analysis; a modern decompiler will first lift assembly instructions into an intermediate representation (IR), which is deemed as more analysis-friendly [16, 59, 61].

*Middle Stage: High-Level Program Recovery.* The ultimate goal of a C decompiler is to convert input executable into high-level source code. Therefore, given the lifted IR code, the central focus is to recover variables, types, and high-level program control flow from low-level IR code. To recover program variables, some tools already proposed and implemented needed static analysis and inference techniques [10, 12, 13, 30, 54]. To recover variable types, constraint-based type inference systems are typically formulated [42, 49].

To recover C-style control structures, modern decompilers implement a set of structure templates and search to determine whether an IR code region matches the predefined patterns. Some advanced techniques enable an iterative refinement to polish the recovered structure. To date, techniques have been designed to guarantee the structure recovery correctness and also to improve readability [17, 67]. In addition, modern decompilers usually design optimizations to polish the lifted IR code, including dead code elimination and untiling [17, 21, 38]. Also, reverse engineering of C executable files might encounter “chicken and egg” problems (e.g., data flow analysis relies on precise output of control flow analysis, and vice versa). Indeed, modules within the middle stage can be invoked back and forth for iterations; the output of one module is used to promote the analysis of other modules [38].

*Back End: Code Generation.* The final stage translates IR statements into C statements and outputs source code. As the output of the middle stage, the IR code is already close to being source code. Translations are mostly mundane (but could still contain bugs, as will show in Sec. 5.1) by concretizing code generation templates.

## 2.2 Equivalence Modulo Inputs (EMI) Testing

We briefly introduce a well-established testing technique that has achieved prominent success in testing compilers, the Equivalence Modulo Inputs (EMI) testing [40]. Soon, we will show that EMI testing can be used to test decompilers in a highly effective manner.

Given a program  $p$  and its legitimate input space as  $\text{dom}(p)$ , the output of  $p$  for an input  $i \in \text{dom}(p)$  is denoted as  $\llbracket p \rrbracket(i)$ . Therefore, two programs  $p$  and  $q$  are defined as equivalent modulo inputs (EMI) in case  $\forall i \in I \llbracket p \rrbracket(i) = \llbracket q \rrbracket(i)$  where  $I \subseteq \text{dom}(p) \cap \text{dom}(q)$ . Here,  $q$  is referred as the EMI variants of  $p$ . The main benefit of EMI testing is the functionality-preservation of  $q$  w.r.t. inputs  $i \in I$ ; therefore, this method does not require a reference implementation and it provides an explicit testing oracle such that any EMI variants  $q$  must behave identically compared with  $p$ . The EMI equivalent property alleviates the notion of “program equivalence.” It consequently provides a viable way to produce programs for testing compilers. To extend the above formulation in testing decompilers, we start by generating  $q$ , which is a mutated program of  $p$  (mutation strategies are introduced in the next paragraph). Then, we decompile the executable file compiled from  $q$  and generate the decompiled source code  $q^*$ . Obviously,  $q^*$  is an EMI variant of  $p$  and we use the above oracle for testing.

The first EMI implementation [40] generates a mutated program  $q$  by profiling  $p$  with inputs  $i \in I$  and deleting or inserting additional statements within uncovered code blocks w.r.t. to any input  $i \in I$ . Since any modified code region in  $q$  is not executed w.r.t.  $I$ ,  $q$  is naturally an EMI variant of  $p$ . Some improvements insert code into the live code region; the inserted code is guarded by opaque predicates that are always evaluated as false during runtime [60]. Some statistical methods are also adopted to optimize the selection of statements for mutations [41]. We employ all three EMI mutation strategies in our new setting (see Sec. 4.1).

## 3 MOTIVATION

In this section, we show the research topic is *central* and *timely*, by shedding light on potential mismatches between traditional

conservative stance of using decompiled C code and the progressive development of decompilation techniques. We also discuss the pitfalls of leveraging IRs for binary code analysis to advocate the development of decompilation techniques.

### 3.1 Research Using C Decompilers

C decompilers are one of the most critical reverse engineering tools that enable various cybersecurity and software re-engineering missions. To date, reusing decompiled x86 and ARM binary code has been a widespread practice, and industry hackers have successfully decompiled and reused complex real-world legacy software, such as video games [1, 3] and complex firmware [4].

In contrast, while decompilers are also extensively used in academia (searching a popular decompiler, “IDA Pro”, in Google Scholar returns 1,370 records since 2015), reusing decompiled C code is not a common approach. Overall, the major application scope of C decompilers in academia includes code comprehension (e.g., similarity analysis) [18, 25, 26], code reuse [27, 37], and various security hardening and vulnerability detection applications [22, 32, 35, 37, 63]. However, our observation is that instead of directly taking final-stage decompiled outputs, intermediate information or representations are usually extracted and leveraged in the conducted research. For instance, instead of directly reusing the decompiled C programs, binary code reuse tasks would be launched with executable patching or replication after reflecting the code layouts and fragments with decompilers [27]. Note that binary patching and replication are usually error-prone and incur a very high cost.

We attribute this conservative and potentially mismatched usage of decompilers to the generally pessimistic views of the decompiled programs: it is traditionally believed that the recovered C code cannot be used for recompilation (unlike conventional C code). Nevertheless, within recent years, promising lines of research have proposed fool-proof techniques for binary disassembling and decompilation [17, 31, 64–67]. As a result, *functionality-preserving* disassembling of non-obfuscated and not-highly-optimized binary code becomes practical, and it could be accurate to assume recovering functionality-preserving C code becomes a matter of engineering effort, although a certain amount of readability of the generated code is sacrificed, e.g., due to the usage of inline assembly or springboard [29, 31]. Indeed, some popular binary analysis frameworks have implemented the proposed techniques [55]: the decompiled output can be smoothly recompiled into an executable that preserves the original semantics [28]. Overall, existing research has shown a practical need to advocate the functionality-preserving “recompilability” as a critical design goal of C decompilers. Given the encouraging development of functionality-preserving reverse engineering, we see this as the perfect time to launch systematic testing of C decompilers, as a means to demystify their full capability.

### 3.2 Analysis and Instrumentation with IR

At present, a decompiler framework often provides an IR lifted from the assembly instructions in support of static analysis and instrumentation. Given the widespread adoption of decompiler IRs for analysis and instrumentation, we argue that using an IR, although not obvious, *does not* primarily eliminate the need for promoting decompilation. In addition to the self-evident reason

<pre> 1. define i32 @bar(i8 par1, i32 par2) { 2.   t1 = alloca i8           type mismatch 3.   t2 = alloca i32 4.   store i8 par1, i8* t1 5.   store i32 par2, i32* t2 6.   t3 = load i8* t1 7.   t4 = sext i8 t3 to i32 8.   t5 = icmp i32 t4, 40 9.   br i1 t5, label1, label3 10. label1:                local vars vs. 11.   ... 12.   br label3             global mem stack 13. label3: 14.   t13 = load i32 t2 15.   ret i32 t13 16. } 17. 18. define i32 @main() { 19.   call i32 @bar(i8 30, i32 60) 20. ... </pre> <p>LLVM IR derived from the sample C code</p>	<pre> 1. t1 = alloca i32 2. t2 = alloca i32 3. store i32 r0, i32* t1 4. store i32 r1, i32* t2 5. //store par1 on stack with offset -5 6. t8 = load i32* t1 7. t10 = load i32** @llvm_fp 8. t11 = getelementptr i32* t10, -5 9. store i32 t8, i32* t11 10. //store par2 on stack with offset -12 11. t12 = load i32* t2 12. t13 = load i32** @llvm_fp 13. t14 = getelementptr i32* t13, -12 14. store i32 t12, i32* t14 15. ... 16. br i1 t18, label1, label3 17. label1: 18. ... 19. br label3 20. label3: </pre> <p>LLVM IR lifted from binary code</p>	<pre> 21. t42 = load i32** @llvm_fp 22. t43 = getelementptr i32* %42 23. ret i8 t45 24. ... 25. %r0 = alloca i32          lack of function boundary info 26. %r1 = alloca i32 27. ... 28. call 840c(i32 t51, i32 t52) </pre> <p>LLVM IR lifted from binary code (cont'd)</p>
		<pre> 1. int bar(char a, int b) { 2.   if (a &gt; 40) 3.     b = b + 10; 4.   else 5.     b = b - 10; 6.   return b; 7. 8. int main() { 9.   int a = bar(30, 60); 10.  return a; 11. } </pre> <p>Sample C code</p>

Figure 2: Comparison between LLVM IR generated from sample C code with IR lifted from binary code compiled from the C code. The sample C code is presented on the lower right corner. LLVM IR code is simplified due to the limited space and the `main` function is backgrounded with grey for better readability.

that many infrastructures and algorithms (e.g., a symbolic execution engine) need to be reimplemented when analyzing customized IRs, performing static analysis and instrumentation regarding a unified IR, e.g., LLVM IR, suffers from similar challenges in variable and type recovery as that encountered in decompilation.

While lifting assembly code into LLVM IR is mostly mundane, the transformed IR lacks high-level expressiveness, and this absence can impede many standard dataflow analyses and symbolic reasoning facilities [10]. In Fig. 2, we study and present a simple example, where a toy C program was compiled into an LLVM IR and further compiled into an executable. We then disassembled the executable into assembly instructions and transformed the instructions into LLVM IR statements. Comparing the two pieces of IR code, we find that much of the high-level program information is missing in IR derived from low-level code, which, as elaborated in [10, 30], will hinder the adoption of many source code-level static analyses due to the missing of program high-level information. Clearly, using IR *does not* primarily eliminate the need for promoting decompilation techniques, since they face the same hurdle regarding type, (local) variable, and control structure recovery.<sup>1</sup> In other words, reverse engineering of high-level data representations are *unavoidable* to facilitate the same amount of analysis expressiveness (although recovery of high-level control structures may not always be needed).

<sup>1</sup>We are certainly aware of some LLVM IR lifters developed by the reverse engineering community [15, 46, 62]. However, experimental tests show that they suffer from similar issues (e.g., lack of local variable recovery), and to date, such lifters are *not* commonly adopted in academia. In fact, researchers prefer to implement their own in-house IR lifter for such tasks [22].

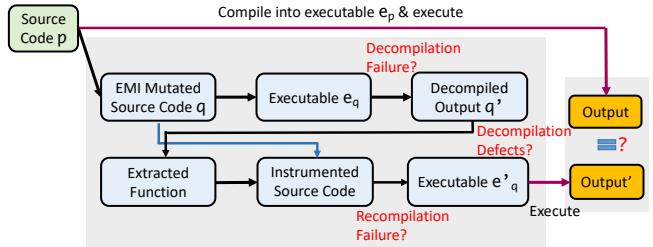


Figure 3: Workflow of our study. Workflow elements ( $p$ ,  $e_q$ ,  $e'_q$ , etc.) are consistent with description in Alg. 1.

Overall, we interpret that analyzing IR code *does not* alleviate need for decompilation, but, instead, could fall into a trap consisting of a “multilingual” scenario where we have to mediate source code-level analysis facilities with low-level code and their incompatible memory models. In summary, our observation again advocates the understanding and development of modern decompilers, which would overcome many common challenges in the first place and enable a seamless integration of source code-level analysis facilities with low-level code study.

## 4 METHODOLOGY AND STUDY SETUP

Fig. 3 depicts the workflow of this empirical study. We start by generating random C programs with a popular generator for compiler testing, Csmith [68]. For each input C code  $p$  generated by Csmith,

we mutate  $p$  with the EMI technique (see Sec. 2.2), and compile the mutated program  $q$  (i.e., an EMI variant) into executable  $e_q$ . The decompiler will take  $e_q$  as the input and produce the decompiled output, another piece of C code. Given that the decompiled C code is not directly recompilable (for the explanation, see Sec. 4.3), we extract the function from the decompiled output (we configure Csmith to generate C code with one function besides `main`) and use this function to replace its corresponding code chunk in  $q$  to generate an instrumented  $q^*$ . Doing so gives us another piece of executable file  $e^*$  compiled from  $q^*$ . To check decompilation correctness, we compare the execution outputs of  $e$  (compiled from the input C code) and  $e^*$ . If a deviant output is identified, we manually inspect and compare  $p$  and  $q^*$  in depth (see Sec. 4.2) and aggregate the harvested information to deduce empirical findings.

**Study Scope.** As depicted in Fig. 3, we capture the following kinds of issues in this study:

- **Decompilation failures** represent errors or crashes thrown by decompilers. This usually indicates a simple decoding bug or ill-format input executables.
- **Recompilation failures** represent errors yielded by the compiler when we are recompiling decompiled outputs. This indicates bugs or implementation limits in decompilers.
- **Decompilation defects** represent deviant results when we execute and compare recompiled executable and its reference input. Deviant outputs imply the semantics of the input executable is broken in the decompiled output, which is likely derived from a decompiler bug.

We aim to reveal and understand various logic bugs or implementation pitfalls that result in incorrect decompiled C code. Therefore, the *recompilation failures* and *decompilation defects* are the major focus (see Table 3 for the findings). While our main focus is not *decompilation failures* since fuzz testing tools could likely find such issues during in-house development, decompilation failures are still recorded and reported in Table 3.

The prerequisite for binary decompilation is disassembling; decompilation is performed to lift the disassembled output into higher-level representations (Sec. 2.1). As mentioned above, while precise disassembling is known to be hard in principle, current algorithms have been shown to work very well in practice and to perform fool-proof disassembling of real-world applications [31, 47, 64, 65]. Therefore, in this study, we assume that binary disassembling is reliable; we focus on analyzing defects in decompilation procedures, where existing research has rarely explored.

We are *not* testing extreme cases to stress decompilers. Decompilers are known to be error-prone for highly optimized and obfuscated code. Indeed, obfuscated and highly-optimized code are not considered by prior “functionality-preserving” disassembling and decompilation research as well [11, 17, 31, 64, 65, 67]. Instead, we aim to understand to what extent modern decompilers are revamped with respect to conventional C programs and provide practical and inspiring insights for decompiler developers and users. We limit our study to unobfuscated and unoptimized binaries compiled on x86 platforms (see discussions regarding other settings in Sec. 6).

---

### Algorithm 1 Decompiler Testing.

---

```

1: function IsDeviant( $p, q$ )
2:    $e_p \leftarrow \text{Compile}(p)$ 
3:    $e_q \leftarrow \text{Compile}(q)$ 
4:    $q' \leftarrow \text{Decompile}(e_q)$ 
5:    $e'_q \leftarrow \text{Compile}(\text{Instrument}(q, q'))$ 
   ▷ Implementation of Instrument is given in Sec. 4.3
6:   if  $\text{Execute\_and\_compare}(e_p, e'_q) == \text{false}$  then
7:     return true
8:   else
9:     return false
10: function Testing( $\mathcal{P}$ )
   ▷  $\mathcal{P}$ : a set of C programs generated by Csmith
11:    $\mathcal{S} \leftarrow \emptyset$ 
12:   for each  $p_k$  in  $\mathcal{P}$  do
13:     if IsDeviant( $p_k, p_k$ ) == true then
14:       add  $(p_k, p_k)$  in  $\mathcal{S}$ 
15:       continue
16:      $q_k \leftarrow p_k$ 
17:     for  $1 \dots \text{MAX\_ITER}$  do
18:        $q_k^* \leftarrow \text{Mutate}(q_k)$ 
19:       if IsDeviant( $p_k, q_k^*$ ) == true then
20:         add  $(p_k, q_k^*)$  in  $\mathcal{S}$ 
21:       if  $\text{Rand}(0, 1) < \mathcal{A}(q_k \rightarrow q_k^*, p_k)$  then
   ▷ Formulation of
    $\mathcal{A}(q_k \rightarrow q_k^*, p_k)$  can be found in [8].
22:          $q_k \leftarrow q_k^*$ 
23:   return  $\mathcal{S}$ 

```

---

## 4.1 Equivalence Modulo Inputs (EMI) Testing

As mentioned above, in this research, we reuse a well-developed compiler testing technique named EMI testing [19, 40, 60]. Recall during decompilation (Sec. 2.1), modern decompilers perform data flow and control flow analyses and transformations, which are conceptually comparable to compiler passes. Therefore, we envision that the full-scale mutations enabled by the EMI technique can adequately expose decompiler defects. Our study confirms this intuition: EMI testing is highly effective in this new setting (for details, see our findings in Sec. 5).

Alg. 1 specifies the workflow of the EMI testing. Function *TESTING* is the main entry point of our algorithm, and *IsDeviant* performs the compilation, instrumentation, and comparison to find deviant outputs of two given programs. In particular, *IsDeviant* compiles input programs  $p$  and  $q$  into two executable files,  $e_p$  and  $e_q$ , respectively; then, it decompiles  $e_q$  into another piece of program  $q'$  (line 4). We then instrument  $q'$  to generate a recompilable program (for the implementation of *Instrument*, see Sec. 4.3) and further compile the code into executable  $e'_q$  (line 5). We then execute these two executable files and compare the execution output (line 6). Note that a program generated by Csmith *does not* require user-provided input; it performs random computations and returns a checksum of its global variables. Therefore, we directly execute programs and compare their outputs, the checksum of global variables.

The input of *TESTING* is a set of arbitrary C programs generated by Csmith, and this function iterates each C program  $p_k$  for the testing (line 12–22). Before performing the EMI mutation, we first use *IsDeviant* to check the recompilation correctness of the seed program  $p_k$  (line 13), and in case a deviant output is found, we record this case (line 14–15) and move to the next program  $p_{k+1}$ .

For each seed program  $p_k$ , we iterate the EMI mutation for  $\text{MAX\_ITER}$  iterations ( $\text{MAX\_ITER}$  is set as 30 in the implementation). At each iteration, we generate a new variant  $q_k^*$  given the

current variant  $q_k$  as the input (line 18). Mutate subsumes mutations of both live code and dead code regions [40, 60]; we identify the live code region of  $p_k$  by executing it and recording the covered statements. While the mutation of dead code regions is mostly straightforward, being performed by inserting and removing unreachable statements, the mutation of live code regions is slightly trickier. Following existing research on mutating live code [60], we insert a set of opaque predicates (we implemented all three opaque predicate schemes proposed in [60]) into the live code region for mutation. When program pairs of deviant outputs are found, we add the pairs into  $\mathcal{S}$  (line 20).

Additionally, instead of the blind mutation strategy (i.e., randomly selecting some statements for mutation) proposed in the first EMI paper [40], we reimplement an advanced EMI mutation strategy guided by the Markov Chain Monte Carlo (MCMC) optimization procedure to explore the search space [41]. For each new EMI variant  $p_k^*$ , we compare it with variant  $p_k$  and compute a program distance, indicating how different they are (line 21). We accept  $p_k^*$  with an acceptance ratio (line 22). This method is designed to generate more diverse EMI mutations progressively since EMI variants with high distance values will lead to a higher chance to be kept [41]. We formulate this MCMC optimization in [8]. After collecting all the program tuples with deviant outputs (line 23), we resort to a manual study to comprehend the root causes of these suspicious outputs.

## 4.2 Manual Inspection

As mentioned above, Csmith-generated C code computes a checksum of its global variables as the execution output. Once a deviant output is identified, we manually pinpoint erroneous statements causing such deviants in the decompiled C code. The manual inspection forms a typical debugging process: starting from a global variable that yields a deviant value, we identify all of its assignments and recursively backtrack the deviant values used for assignments until we identify the root cause (i.e., the erroneous statements) in the decompiled EMI variants.

This step is costly: examining and checking all suspicious findings took two reverse engineering analysts about 180 man-hours. Each analyst has an in-depth knowledge of reverse engineering and rich experience in binary code analysis and CTF competitions. In this way, we ensure the accuracy of our study and the credibility of our findings to a great extent. We have reported our findings to the developers, and some typical cases have been promptly confirmed and to be fixed (see Sec. 5). Moreover, with about 350 man-hours, we locate 13 buggy code fragments in open-source decompilers that lead to erroneous statements in the decompiled C code (see discussions in Sec. 5.1).

## 4.3 Study Setup

The proposed study is implemented in Python, with 3,707 LOC (measured by cloc [23]). We now discuss challenges and practical solutions involved in setting up this study.

**Challenges for Recompilation.** The standard EMI technique employs a straightforward testing oracle by comparing execution results of a seed program and its EMI variants. However, shortly our

findings will show that it is difficult to directly recompile the outputs of decompilers (see Sec. 5.2.2 for the details). Our study shows that many *undefined* symbols (e.g., ELF binary-specific symbols) commonly exist in the global data sections of the decompiled outputs. Below, we discuss solutions to address this issue.

**Implementation of *Instrument*.** We acknowledge the difficulty of recompiling the decompilation outputs; to enable an automated workflow for testing and empirical study, we seek to extract functions from the decompiled C code and use the decompiled functions to replace their corresponding code chunk in the source code of the decompiler input (see “Instrumented Source Code” in Fig. 3). We then execute the Csmith-generated C code and its instrumented EMI variant and check for any execution result deviation.

The intuition is that typically within each decompiled function, symbols (e.g., local variables) are defined in a complete manner, and therefore, the whole chunk becomes a “closure.” In fact, as just mentioned, the undefined symbols are mostly placed in global data sections, and based on our observation, they usually do not interfere with computations within each function. To perform this extraction, we configure Csmith by bounding the maximum number of functions in its output as one (in addition to `main`). As will be reported in our findings, this method revives the “recompilability” of modern decompilers when processing most C programs.

**Handling Discarded Global Variable Names.** Names of most variables are discarded after compilation, and in the decompiled outputs, variables are renamed meaninglessly (`v0`, `v1`, etc.). Although doing so does not affect the usage of local variables, this study requires to correctly resort the usage of global variables, since global variables are frequently referred by Csmith-generated code (recall that the execution result of a Csmith-generated program is the checksum of its global variables).

We address this issue by creating a ghost local variable for each global variable, and replace the usage of global variables with their corresponding local variables. Consider the example below:

```
int ga;
void set_var(int la){
    // synchronize global var. with local var.
    ga = la;
}
void foo(){ // Csmith generated function
    int la;
    ... // usage of ga are replaced by la
    set_var(la);
    // compute checksum of ga and print output
}
```

After compiling and decompiling, the local variable name (`la`) is discarded; however, global variable `ga` will be updated before exiting `foo` since its corresponding local variable in the decompiled code is still kept as the parameter of `set_var`. We configure Csmith to avoid the usage of C pointers, and therefore, we do not need to resolve alias issues when identifying the usage of global variables.

**Configurations.** We use Csmith (ver. 2.3.0) [68] to generate seed C programs. gcov (ver. 7.4.0) is employed to obtain code coverage

information and to identify live code for EMI mutations. We compile Csmith’s outputs with gcc (ver. 7.4.0) into 32-bit x86 binary code with no optimization.

Tentative studies show that seed C programs with complex data structures impede recompilation. Therefore, in this study (except Sec. 5.2.2), we configure Csmith and ensure that its outputs contain only a subset of C grammars. We exclude C struct, union, array, and floating points. As mentioned in this section, we disable pointers to simplify the manual study and avoid alias analysis when creating ghost local variables for global variables. Note that while this configuration simplifies data structures, Csmith still generates programs with *highly complex* control structures, arithmetic operations, and type castings.

Intuitively, decompilation problems are more obvious for large and complex software. However, we note that using large software, while could likely provoke errors, is not feasible in this research. As mentioned in Sec. 4.2, we manually inspect every erroneous decompiled code to understand which statement is decompiled wrongly. This manual inspection already takes about 180 man-hours. Errors in complex and large software could make the manual inspection too costly or even infeasible. Nevertheless, our findings on real-world decompilers are general enough to affect the decompilation of any C code and therefore fixing our findings will presumably promote the decompilation of large and complex C software.

**Decompilers.** Table 1 reports the decompilers used in our study. IDA-Pro [34] and JEB3 [52] are both the state-of-the-art commercial decompilers that have been widely used in many research and industry projects. To strengthen the generalizability of our study, we also evaluate RetDec [38] and Radare2 [2], two popular free decompilers that are actively maintained by the community. The implementation details of commercial decompilers are mostly obscure to the public. In contrast, RetDec shares a very promising vision to bridge binary code analysis with the LLVM ecosystem. It is built as a reverse engineering frontend of the LLVM framework, and users are allowed to implement their analysis and instrumentation passes using LLVM. Radare2 recently integrates the Ghidra decompiler [50] developed by NSA. The Ghidra plugin leverages the front-end of Radare2 for disassembling, and use the Ghidra decompiler (version 9.1) to convert the disassembled code into C code. Our tentative test shows that Ghidra plugin has much better decompilation accuracy than the native decompilation support of Radare2 which is still immature.

These four decompilers, to the best of our knowledge and experience, represent the best two commercial and best two non-commercial C decompilers. We indeed tentatively explored other decompilers in our study. For instance, Snowman [5] is seen to produce worse decompiled outputs compared with these four decompilers, and there is no manual provided [7].

Decompilers are generally designed for an “out-of-the-box” usage. We cannot find options to configure optimizations (not like compilers) or decompiling algorithms. We also cannot find documents shipped with these decompilers on how to configure them. Hence, all decompilers are studied in the standard setting.

**Statistics of Test Cases.** Table 2 reports statistics of the C programs used in the study. We use Csmith to randomly generate

**Table 1: Decompilers employed in the study.**

Tool Name	Information
IDA-Pro [34]	Commercial
JEB3 [52]	Commercial
RetDec [38]	Free; maintained by the community
Radare2/Ghidra [2, 50]	Free; maintained by the community and NSA

**Table 2: Statistics of the C programs used in the study. Line of code (LOC) is measured with cloc [23].**

Total # of programs generated by Csmith	1,000
Total LOC in Csmith generated C programs	142,888
Total # of EMI variants	9,707
Total LOC in EMI variants	2,361,590

1,000 C programs. These 1,000 programs are the seed inputs of EMI mutation for each decompiler. The total number of generated EMI variants is 9,707 (see Table 3 for the breakdown). As mentioned in this section, we configured Csmith to produce one function (in addition to main) for each C code it generates. Each seed program (on average 148 LOC) contains a medium size function with presumably complex control structures and many global variables.

## 5 FINDINGS

Table 3 provides an overview of our findings. Out of in total 9,706 test cases (exclude one decompilation failure), we find 408 (4.2%) recompilation failures. While most decompiled C code can be successfully recompiled and executed, 1,014 (430+584; 10.4%) outputs are erroneous: the decompiled C code shows deviant execution results compared with the seed. JEB3 outperforms the other three decompilers, given less decompilation defects (30+9). Despite one decompilation failure in JEB3 (errors thrown by the decompiler), all the cases can be decompiled smoothly.

We further put these decompilation defects into five categories (under the “Characteristics” column) by identifying and classifying erroneous statements in decompiled C code. As mentioned in Sec. 4.2, we form a group of reverse engineering analysts to classify the findings manually. This ensures the accuracy of our research. Nevertheless, we admit the difficulty, given the large number and highly-optimized decompiled C code (see Sec. 5.3.4 for corresponding discussion). Therefore, we do classification at our *best effort*. The first three columns stand for errors (in total 861) found in types, variables, and control flow structures of decompiled C code. We also find 147 errors that are presumably due to decompiler optimizations. “Others” report 6 errors that are likely due to bugs in the IR-to-C translation stage. We give further discussions and case study regarding each category in Sec. 5.3.

**Processing Time.** Our experiments are launched on a machine with Intel Core i5-8500 3.00 Hz CPU and 4 GB memory. Although processing time is in general not a concern for decompilation, we record and report that it takes on average 5.0 CPU seconds to decompile one case. We interpret that de facto decompilers perform efficiently in processing commonly-used binary code.

**Confirmation with the Decompiler Developers.** We have reported all the failures and defects found in this research (findings

**Table 3: Result overview.** As depicted in Alg. 1, if a Csmith generated program  $p_0$  has inconsistent functionality compared with its decompiled output  $p_0^*$ , we skip the EMI mutation on  $p_0$  and directly report it as one decompilation defect (the “Csmith Output” subcolumn). Otherwise, we generate 30 EMI mutations from each  $p_0$  as EMI testing inputs. The total number of EMI mutations are reported in “# of EMI Variants.” We manually analyzed each defect and summarized findings in “Characteristics”.

Tool Name	# of EMI Variants	Decompilation Failures	Recompilation Failures	Decompilation Defects		Characteristics of Decompilation Defects				
				Csmith Output	EMI Variant	Type Recovery	Variable Recovery	Control-Flow Recovery	Optimization	Others
IDA-Pro	3,786	0	208	1	69	2	0	4	61	3
JEB3	2,510	1	13	30	9	25	7	0	4	3
RetDec	907	0	187	346	380	315	338	25	48	0
Radare2/Ghidra	2,504	0	0	53	126	35	87	23	34	0
Total	9,707	1	408	430	584	377	432	52	147	6

in Table 3) to the decompiler developers. To seek for prompt confirmation and insights into our results, we also select at least one example in each defect category and present to the developers.

The JEB3 developer was responsive in confirming the selected cases for each defect category. He even mentioned to tentatively include some findings in the upcoming major update. To quote him:

“thanks so much for all that feedback!! - next update ...  
I hope to include some of your reports! ...”

We also quote the IDA-Pro author’s feedback below:

“We internally use Csmith to test the decompiler and we know that our decompiler can not handle all cases yet. We are working on them.”

We interpret that the IDA-Pro developers certainly care about the *functional correctness* of decompilation. At the time of writing, we are waiting for the response from RetDec and Radare2/Ghidra. Overall, we understand that these decompilers are developed by either small companies (like the commercial ones) or volunteers, and it certainly takes a while for confirmation and to incorporate our findings into the scheduled development pipeline. Nevertheless, responses from IDA-Pro and JEB3 developers indicate that they take our findings seriously.

## 5.1 Identify Buggy Code Fragments in Decompilers

We seek to pinpoint the buggy code fragments in decompilers that lead to these defects. IDA-Pro and JEB3 are commercial tools and therefore we have no way of performing root cause analysis. Radare2/Ghidra and RetDec both have source code available online, although their accompanying documents and code comments are merely provided. At this step, we spent *extensive* manual efforts (over six weeks; in total about 350 man-hours) to analyze all 913 (187 + 346 + 380) decompiler flaws detected from RetDec (in total 178,732 LOC), and all 179 (53 + 126) decompiler flaws in Radare2/Ghidra (the Ghidra plugin has 112,999 LOC). As reported in Table 4, 13 buggy code fragments are found from these two decompilers.<sup>2</sup> To clarify potential confusions on Table 4, recall to decide the “characteristics” of decompilation defects, we manually analyzed decompiled C code w.r.t. its input seed. We summarized erroneous code statements into five categories at our best effort

**Table 4: Buggy code fragments found in RetDec (3rd to 10th rows) and Radare2/Ghidra (11th to 17th rows).** Column names, simplified due to the limited space, shall be easily figured by referring to Table 3.

	Recompile Failure	Decompilation Defects				Total
		Type	Var.	Control.	Opt.	
Bug1	0	297	0	0	0	297
Bug2	0	0	338	0	0	338
Bug3	0	0	0	11	0	11
Bug4	0	18	0	14	4	36
Bug5	0	0	0	0	44	44
Bug6	177	0	0	0	0	177
Bug7	10	0	0	0	0	10
<b>Total</b>	<b>187</b>	<b>315</b>	<b>338</b>	<b>25</b>	<b>48</b>	<b>913</b>
Bug8	0	31	87	0	0	118
Bug9	0	0	0	6	10	16
Bug10	0	0	0	3	3	6
Bug11	0	1	0	5	0	6
Bug12	0	0	0	5	21	26
Bug13	0	3	0	4	0	7
<b>Total</b>	<b>0</b>	<b>35</b>	<b>87</b>	<b>23</b>	<b>34</b>	<b>179</b>

in Table 3. Nevertheless, with root cause analysis, we find that decompiler bugs may cause different errors in decompiled code. For instance, Bug8 is a type recovery bug in Radare2/Ghidra, directly generating 31 decompiled C programs with type errors. Furthermore, given local variables of wrong types, optimization may treat these variables as part of other variables (see Sec. 5.3.2), outputting 87 decompiled C programs with missing variable errors.

All of these findings are logic bugs, causing erroneous outputs rather than decompiler crash or abnormal termination. From these 13 bugs, 12 are found from the “middle stage” of decompilation (Sec. 2.1) where decompilers perform high-level program representation recovery, while one (Bug13 found in Radare2/Ghidra) is in the C code generation phase: unsigned shift operation in the Ghidra IR was lifted into a signed shift in C code. From the 12 middle stage bugs, four bugs are in the type recovery modules, two bugs are in the variable recovery modules, and six bugs are in the optimization modules. Also, while Radare2/Ghidra is stated to leverage the disassembly infrastructure of Radare2 and bridge with Ghidra, we find that Radare2/Ghidra “freerides” the type recovery utility of Radare2. Bug8 in Radare2/Ghidra indeed roots from incorrectly recovered variable types in Radare2.

## 5.2 Recompilation Study

We start by answering **RQ1: how difficult is it to recompile the outputs of modern decompilers?** To that end, we conduct a two-step study

<sup>2</sup>This section reports and discusses high-level information in the main paper. Information regarding each buggy code fragment, including their locations, descriptions and samples erroneous outputs they could incur, can be found in [8].

in this section. We first evaluate the recompilation of decompilers by directly compiling their outputs. Given the general challenge in the first step, we then resort to instrument the decompiled outputs (with *Instrument* defined in Sec. 4.3) and analyze the remaining recompilation failures reported in Table 3.

**5.2.1 Full-Scale Recompilation.** We first study full-scale recompilation by directly recompiling the recovered high-level C code of de facto decompilers. To do so, we use the default option of Csmith to randomly generate 100 C programs and feed to the decompilers.<sup>3</sup> We then recompile their outputs and collect compiler messages.

We report that we are unable to recompile any of the decompiled outputs into functional executables. Plenty of *undefined* symbols exist in the decompiled outputs, including ELF binary specific symbols (e.g., symbols used for dynamic linkage), decompiler specific symbols or undefined variables. In short, we interpret that outputs of de facto decompilers are not directly recompilable, and more importantly, by putting decompiler and executable specific symbols in their outputs, recompilation seems not the top priority for de facto decompilers (although “readability” in the decompiled outputs is also not well supported, as we will show in Sec. 5.3.4).

Therefore, in the rest of the section, we follow what has been proposed in our study setup (Sec. 4.3) and discuss the recompilation failures of the instrumented decompilation outputs. We also present a corresponding discussion below in Sec. 5.2.3.

**5.2.2 Recompilation Failure.** We now discuss the causes of the recompilation failures reported in Table 3. We manually checked the compiler errors for the 408 recompilation failures and identified two key reasons that impede recompilation.

**Erroneous Variable Recovery.** We find many variable recovery errors in the decompiled outputs. For instance, when decompiling a C program with only one local variable with JEB3, the output is translated into the following statement:

```
// source code
unsigned int a;
```

```
// decompiled code
unsigned int v0, v0;
```

Despite the difficulty to locate the buggy component in JEB3 that causes such re-declaration issue, we report that similar problems were found in a considerable number of JEB3’s outputs.

We also found recompilation failures due to incorrect recovery of function call parameters (found in JEB3 and RetDec). Consider the example below:

```
// source code
int a = 12;
int b = 10;
set_var(a,b);
```

```
// decompiled code
int a = 12;
int b = 10;
set_var(a);
```

where the decompiled output causes an inconsistency in the declaration and invocation of function `set_var`. In fact, manual inspection in Sec. 5.1 shows that all 187 recompilation failures of RetDec root from two bugs (i.e., Bug<sub>6</sub> and Bug<sub>7</sub> in Table 4) in the function prototype recovery module. In other words, fixing these two bugs of function call parameter recovery would eliminate all 187 failures.

<sup>3</sup>To clarify potential confusions, these 100 C programs are only used as a quick check on recompilation at this step. All the other testing and studies are from a set of 1,000 C programs, as explained in the study setup (Sec. 4.3).

**Undefined Symbols.** Despite that undefined symbols are mostly eliminated in this new setting, still, such issues can be found in IDA-Pro’s outputs. In particular, we report that ELF binary specific symbols (e.g., symbols of Global Offset Table used for dynamic linkage) seem to be placed in outputs of IDA-Pro commonly. In contrast, the other three decompilers work generally well to avoid the abuse of undefined symbols.

**5.2.3 Result Implication.** Our study on full-scale recompilation (Sec. 5.2.2) shows that outputs of decompilers are still not directly recompilable. However, one previously-ignored fact revealed in this study is that generating recompilable code of many non-trivial C programs is essentially the *last mile* of modern decompilers. As shown in this research, after some *syntax-level* tweaks, most decompiled outputs become recompilable. Radare2/Ghidra shows highly encouraging findings with zero recompilation failure, and both commercial decompilers have less than 5.0% recompilation failure. IDA-Pro fails 208 cases by inserting extra ELF binary specific symbols, most of which are used for dynamic linkage. Our study further indicates the possibility to safely remove some of them since the linker will need to create these symbols in the ELF executable files when recompiling. Also, while RetDec has 187 recompilation failures, our root cause analysis shows that these errors are due to only two bugs in the function parameter recovery module, which shall be fixed together smoothly.

Although academic researchers were generally believing that outputs of C decompilers are not for reuse and not even recompilable, this study re-scopes this pessimistic stance by demonstrating that after some systematic and straightforward syntax-level changes (without any manual tweaks), modern C decompilers show decent capability of recompiling non-trivial C programs (recall our C programs have relatively simple data structures but *complex* control structures, arithmetic operations, and type castings). Therefore, we summarize and present our first finding as follows:

**Finding:** By applying straightforward syntax-level changes without any manual tweaks, modern C decompilers already show good support of recompilation for many non-trivial C programs.

Existing research has shown a practical need to advocate the “recompilability” as a critical design goal, if not foremost, of decompilers. This study shows that it requires only syntax-level changes to revive the decompiled outputs of many non-trivial C codes. The recompilability can drastically promote the reuse of legacy software, and becomes the “next big thing” in the community. Indeed, starting from a piece of reassemblable assembly code by reusing existing research tools [31, 64, 65], we envision the feasibility to deliberately craft decompilation passes and deliver recompilability-preserving decompilation, by making readability a secondary consideration (e.g., preserving certain onerous instructions with inline assembly) or framing it in terms of good faith effort.

### 5.3 Decomposition Defects

This section answers RQ2: *what are the characteristics of typical decompilation defects?* Table 3 classifies all the deviant outputs into five categories. In the rest of this section, we elaborate on typical errors within each category.

**5.3.1 Type Recovery.** The lack of fool-proof type recovery is one key limit of existing decompilers. While for C code, not all the type recovery failure breaks the semantics, in this study we flag a considerable amount of recovered variable types breaking semantics (in terms of both control and data flow). Consider an example below:

```
// source code
int i = 0;
for(i=6; i<-12; i-=6){
    ... // not reachable
}
```

```
// decompiled code
unsigned int i = 0;
for(i=6; i<-12; i-=6){
    ... // reachable
}
```

where the Csmith-generated C code has a `for` loop, but will execute for zero iterations. Contrarily, the type recovery incorrectly annotates `i` with `unsigned int` in the decompiled C code, and the loop body becomes reachable in the decompiled output. In other words, the type recovery alters the *control flow*, and further changes the semantics.

We also find a considerable amount of type recovery failure that leads to an erroneous *data flow*. Consider a case below found:

```
// source code
int32_t a = 0xaaffff;
```

```
// decompiled code
int16_t a = 0xaaffff;
```

In the decompiled output, the variable type `int32_t` is incorrectly recovered as `int16_t`. As a result, `a` is initialized with a different value, since a variable of `int16_t` type can only take the lowest 16 bits (0xffff).

As reported in Table 4, 315 type errors in C code decompiled by RetDec root from two bugs (Bug<sub>1</sub> and Bug<sub>4</sub>). Bug<sub>1</sub> happens by converting signed mov statements (`movsx`) in x86 assembly into IR statements without correctly considering extensions and truncations. Bug<sub>4</sub> missed certain function call parameters when analyzing the call site stack push operations. Bug<sub>8</sub>, Bug<sub>11</sub>, and Bug<sub>13</sub> cause in total 35 type errors in Radare2/Ghidra. Bug<sub>8</sub> cannot precisely infer the “length” of stack variables, while Bug<sub>11</sub> performs constant propagation optimization and uses a ghost variable defined in Ghidra of wrong type to replace original variables. As aforementioned, Bug<sub>13</sub> lifts unsigned shift in IR into signed shift in C code.

**5.3.2 Variable Recovery.** We find 338 variable recovery issues from RetDec, where this decompiler seems unable to recognize certain variables and future leads to deviant execution outputs compared with the seed programs. Root cause analysis shows that Bug<sub>2</sub>, a bug on function parameter recovery, incurs all these 338 errors. Radare2/Ghidra can also miss to identify certain local variables. Root cause analysis shows that due to bugs in type recovery module of Radare2 (i.e., Bug<sub>8</sub>), variable is incorrectly assigned with a larger size (e.g., a 32-bit integer). Variables adjacent to this “larger” variable can overlap in the memory layout, and can be potentially deemed as part of this “larger” variable and is therefore optimized out.

We also find seven erroneous variable recoveries in JEB3 which causes “undefined behavior” in the decompiled C code. Consider a simplified case:

```
unsigned int v0;
unsigned int v1 = v0 >> 16;
```

where `v0` is defined yet uninitialized, leading to an indeterminate value in `v1`. We have confirmed that the source code does not contain any uninitialized local variables. In general, generating code with undefined behavior is undesired for decompilers. Although

the root cause is yet to be determined, we suspect that `v0` in the above case is hardcoded in the output. We urge the decompiler developers to avoid generating code exhibiting undefined behavior, for instance, by initializing `v0` with zero.

**5.3.3 Control-Flow Recovery.** We find a total of 52 deviant outputs due to control-flow errors in the decompiled C code. We note that after taking a close look at the buggy code fragments, *all* failures in RetDec and Radare2/Ghidra are actually due to wrong type recovery and optimization bugs. Consider a simplified input program below:

```
// condition generated by EMI
if (opaque_condition){ // evaluated to false
    statement1; // not reachable
}
```

where the `opaque_condition` will be evaluated to false during runtime. However, we report that the `if` condition was optimized out in the decompiled program since the condition is incorrectly evaluated to “true” due to type recovery or optimization errors. Hence, the unreachable branch becomes reachable, reflecting a “control-structure” error in the decompiled C code. We report that Bug<sub>3–4</sub> found in RetDec and Bug<sub>9–13</sub> in Radare2/Ghidra all lead to such issues, although they represent different buggy code fragments in the type recovery and optimization modules.

We also find issues where the statements are mistakenly reordered in IDA-Pro outputs. Consider the case below:

```
// source code
if (cond1)
    statement1;
if (cond2)
    statement2;
```

```
// decompiled code
if (cond2)
    statement2;
if (cond1)
    statement1;
```

where in the decompiled outputs, two `if` statements are reordered, altering execution flow, and leading to deviant execution outputs.

**5.3.4 Optimization.** Academic researchers design decompiler optimizations to make decompiled code close to the input source code and thus more “readable” [17, 67]. However, optimizations, particularly the erroneous constant folding and constant propagation, can make decompiled C code mal-functional. As aforementioned, from the 13 bugs reported in Table 4, six are within the optimization modules of decompilers, for instance performing constant folding without correctly taking bit length into account (Bug<sub>5</sub>), or incorrectly using a 32-bit ghost variable defined in Ghidra to replace a 16-bit variable during constant propagation (Bug<sub>11</sub>).

In addition to errors, we note that the overly (and wrongly) optimized C code often becomes *too concise*. Such aggressive optimization has diminished the readability of decompiled C code to a great extent, and has caused a major challenge for our manual inspection (Sec. 5.1). While “readability” could be a subjective criterion, inspired by our observation, we measure the readability by 1) reporting the average LOC for the decompiled code, and 2) analyzing the code similarity between the decompiled outputs and the input programs. We leverage a popular software similarity analyzer, moss [6], to measure code similarity. The similarity score (ranging from 0 to 1.0; higher is better) indicates how close the decompiled outputs and inputs are. The results are reported as follows:

	<b>IDA-Pro</b>	<b>JEB3</b>	<b>RetDec</b>	<b>Radare2/Ghidra</b>
LOC	90	85	54	104
Similarity Score	0.51	0.50	0.54	0.49

Decompiled programs are highly concise, in the sense that their average LOC is much lower than the corresponding LOC of input programs (on average 143; see Table 2). Even worse, the average similarity score between decompiled outputs and input programs is also low. In summary, we interpret that the highly-optimized decompiled outputs obstruct the readability notably (see further discussions in Sec. 5.5).

Also, the above results may (incorrectly) indicate that decompilers “consistently” optimize their outputs, since their average similarity scores w.r.t. reference inputs are close. Nevertheless, Sec. 6 will show that indeed for an input executable, its corresponding outputs of four decompilers have very different representations, indicating a strong need for regulating optimization and code generation.

## 5.4 Others

We find six errors in the “Others” category: we report that the decompiled outputs have syntax-level difference (e.g., arithmetic operators) compared with the input source code. While the root cause is unknown (since these two decompilers are closed source), we suspect such issues are due to sloppy errors in the C statement translation stage, which could be fixed by developers easily.

## 5.5 Result Implication

In this section, we present discussion and result implication to answer research question **RQ3: what insights can we deduce from analyzing the decompilation defects?**

**Support of Cutting-Edge Research Outputs.** We have found plenty of type, variable, and control structure errors in the decompiled C code, and explained their corresponding buggy fragments in the decompilers. Although academic researchers are believed to have mostly addressed such reverse engineering challenges (since we are evaluating non-trivial but *not extreme* cases), one observation is that modern decompilers have not fully implemented those well-established research products. For instance, while the state-of-the-art research has been working on function prototype recovery for years and achieved promising results (e.g., close to 99% accuracy for function recognition in x86 binaries [14, 20, 58]), still, the de facto decompilers (e.g., RetDec) have not implemented the proposed methods and therefore make lots of errors in recovering function prototypes and parameters. Also, recovering types from x86 binary code have been formulated as a recursively-constrained type inference approach with sub-typing, recursive types, and polymorphism [42, 49]. Contrarily, C decompilers, from the disclosed documents and our observation, only implement simple inference techniques combined with heuristics and predefined patterns [33, 38, 53].

**Finding:** The de facto decompilers still have not fully leveraged the research outputs in this field to improve reverse engineering accuracy.

Although research products cannot be used to address every corner case, most defects exposed in this study, such as RetDec’s obvious limit in recovering function prototypes, shall be fixed smoothly.

Overall, while modern decompilers perform decently in recovering high-level source code, we urge developers to embrace research products in this field to revamp the design and solve problems exposed in this study.

**Optimization.** As disclosed in Sec. 5.3.4, our study on de facto decompilers uncovers the following finding:

**Finding:** De facto decompilers extensively simplify their outputs, even though readability and decompilation correctness are undermined simultaneously.

We encountered major difficulty when manually inspecting erroneous outputs that are highly simplified. We suspect that if it was not even presentable for us — reverse engineering analysts — to comprehend the decompiled outputs, it should be accurate to assume the outputs are often not readable enough for layman users. This clearly indicates a *mismatch* between expectations in the literature and the actual capabilities of modern decompilers. Note that in academia, the optimization module of decompilers is designed following the principle of *enhancing* the readability and making it close to the original C code [17, 67].

Although optimization can help to simplify code emitted by decompilation passes and can usually output one succinct high-level statement by folding several statements, we advocate fine-grained calibration. Currently, modern decompilers seem to go too far and notably hurt readability of their outputs. Meanwhile, motivated by how compiler optimizations are provided for usage, we urge decompiler developers to make their products configurable to flexibly select optimization passes.

## 6 DISCUSSION

**Limitations and Threat to Validity.** We now give a discussion of validity and shortcomings of this paper’s approach. In this research, *construct validity* denotes the degree to which our metrics actually reflect the correctness of C decompilers. Overall, we conduct dynamic testing and manual inspection to study the outputs of de facto decompilers. Hence, while this practical approach detects decompiler bugs and reveals inspiring findings, the most possible threat is that our testing approach cannot guarantee the functional correctness of decompilers. We clarify that our work roots the same assumption as previous works in this line of research that aim to comprehend the functionality of reverse engineering toolchains with dynamic testing rather than static verification [36, 51].

We check the correctness of decompiled C code by comparing its execution output with its reference input program. Considering the execution output of each Csmith generated program is a checksum of all its global variables, decompilation errors on global data or its involved computations can be faithfully exposed. However, a possible threat is that defects can be neglected in the decompiled C code, in case they do not contribute to the execution output. One promising mitigation is to enable a static viewpoint of the decompiled output. Instead of executing the decompiled code, we envision opportunities to perform whole-program comparison and pinpoint inconsistency. We leave exploring this direction for future work.

Besides, there exists the potential threat that the proposed decompiler testing framework may not adapt to other types of programs,

since the conducted research focuses on C code decompilation. Nevertheless, we mitigate this threat to *external validity* by designing an approach that is language and platform independent. As a result, our approach is applicable to other settings outside the current scope. We believe the proposed technique is general, and we give further discussions regarding other decompilation settings soon in this section.

**Decompiler Developers’ Responsibility.** We consider that developers should take the responsibility to constructively address the findings in this research. Our work serves as the first and systematic effort to provide guidelines. In Sec. 5.5, our findings have shown that modern decompilers still have not leveraged the full potential of research outputs. Our research sheds light on where developers can start to enhance their products, e.g., avoiding extensively simplifying the decompiled C code (see Sec. 5.3.4).

Looking ahead, we also advocate decompiler developers to embrace breakthroughs of “semantics-preserving” reverse engineering [17, 31, 47, 64, 65]. Therefore, decompiled outputs could become fool-proof “recompilable” in the first place. We also envision the need to deliver more principled techniques to verify the functional correctness of decompilation. Meeting this need will have a prominent, long-term impact in the reverse engineering community.

**Cross Comparison of Decompiled C Code.** Careful readers may wonder the feasibility of conducting a static “cross comparison”, by decompiling the same executable with a set of decompilers and identifying differences in their outputs. However, we note that decompiled C code can have drastically different representations since different decompilers implement their own tactics and translation templates (although they share identical semantics). Here, we compile 500 programs randomly generated by Csmith into executable files. For each executable file, we use four decompilers to decompile it and cross compare the similarity (also with moss [6]) of four decompiled C code. We report the average similarity score as follows:

	IDA-Pro	JEB3	RetDec	Radare2/Ghidra
IDA-Pro	×	0.73	0.69	0.66
JEB3	×	×	0.68	0.65
RetDec	×	×	×	0.68
Radare2/Ghidra	×	×	×	×

The average cross similarity score is indeed low (on average 0.68), which sheds light on practical needs to advocate more consistent representations and regulations. Overall, we leave it as one future work to explore practical methods to perform cross comparison, for instance by extracting certain “semantics-level” invariants.

**Other Settings.** The main focus of this study is C decompilation, one challenging and fundamental task commonly encountered in real-world cybersecurity and software re-engineering missions. While the current experiments is conducted on x86 platforms, given popular decompilers like IDA-Pro and RetDec can handle different processors and formats (e.g., ARM and NIPS), we envision opportunities for the research community to generalize our findings, since the key issues, including both recompilation and decompilation defects, are mostly platform and language *independent*.

Decompiling bytecode (e.g., Android apps) is easier, and the quality of decompiled code is generally deemed as higher. Indeed, the Android repackaging attack has become an “out-of-the-box” practice, for which correctly decompiling bytecode is the pre-requisite. In contrast, decompiling non-trivial C++ code, for instance recovering its class hierarchy, is still an open problem [56]. We leave it as one further work, to generalize methodologies proposed in this work on studying other popular decompilation settings.

## 7 RELATED WORK

Testing techniques have been used to measure *static reverse engineering* tools. For instance, differential testing has been used to validate disassemblers [51]. Recent research uses symbolic equivalence checks (with symbolic execution and constraint solving) to pinpoint bugs in IR lifters of binary executables [24, 36]. The security community has also conducted remarkable empirical studies regarding the accuracy and usage scenarios of de facto disassemblers [11].

State-of-the-art *dynamic reverse engineering* activities mostly use symbolic execution techniques and virtual machine (VM)-based monitoring to capture abnormal and malicious behaviors of suspicious binary code. Existing research has proposed various testing techniques to examine the security and reliability of these dynamic reverse engineering tools. Red pill testing [44, 45, 57] leverages the random or differential testing (typically with a black-box setting) to compare the behavior of a VM and that of a physical machine when executing with the same input. Recent research has promoted the testing of a low-confidential emulator with inputs generated by analyzing a highly confidential emulator [43].

Existing research has laid a solid foundation on testing static and dynamic reverse engineering tools. However, a thorough and complete testing of decompilers is still the missing piece in the understanding of today’s reverse engineering landscape. There is a demanding need to gain insights into how much of a problem decompilation is, given its indispensable role in building cybersecurity and software reuse applications.

## 8 CONCLUSION

We have performed a systematic study to investigate decompilation correctness of modern C code decompilers. Our large-scale evaluation on four popular commercial and free decompilers successfully found considerable decompiler flaws. In addition, we elaborately explained findings and summarized lessons we have learned from this study. We show that modern C decompilers have been progressively improved to generate quality outputs. Nevertheless, some classic reverse engineering challenges, including type recovery and optimization, still frequently impede modern decompilers from generating well-formed outputs. This work could provide guides for researchers and industry hackers that aim to use and improve C decompilers, and is presumably adaptable to test other decompilation settings.

## ACKNOWLEDGMENTS

We thank the anonymous ISSTA reviewers for their valuable feedback. Our special thanks go to the JEB3 and IDA-Pro developers who provided us with much help, insight and advice.

## REFERENCES

- [1] 2014. Starcraft Reverse Engineered to run on ARM. <https://news.ycombinator.com/item?id=7372414>.
- [2] 2016. radare2. <http://www.radare.org/r/>.
- [3] 2018. Diablo devolved - magic behind the 1996 computer game. <https://github.com/diasurgical/devilution>.
- [4] 2018. Firmware Mod Kit. <https://github.com/rampageX/firmware-mod-kit>.
- [5] 2018. Snowman decompiler. <https://derevenets.com>.
- [6] 2019. Moss: A System for Detecting Software Similarity. <https://theory.stanford.edu/~aiken/moss>.
- [7] 2019. Output of nocode Invalid C++ Code. [https://github.com/yegord/snowman\\_issues/196](https://github.com/yegord/snowman_issues/196).
- [8] 2020. Decompiler Flaws and Root Cause Analysis. <https://www.dropbox.com/sh/kqw7e19snfeukai/AADHZ45TAL9Kxi7v9nmxdxfLca?dl=0>.
- [9] 2020. Decompiler Fuzzing Test with EMI mutation. <https://github.com/monkbhai/DecFuzzer>.
- [10] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *EuroSys '13*.
- [11] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Sec.*
- [12] Gogul Balakrishnan and Thomas Reps. [n.d.]. DIVINE: DIIscovering Variables IN Executables. In *VMCAI 2007*.
- [13] Gogul Balakrishnan and Thomas Reps. 2010. WYSIWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug, 2010), 84 pages.
- [14] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association.
- [15] Ahmed Bougacha. 2016. Dagger. <https://github.com/repzret/dagger>.
- [16] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform (*CAV*).
- [17] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 353–368.
- [18] Mahinthan Chandramohan, Jinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS Binary Search (*FSE*).
- [19] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *PLDI*.
- [20] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 99–116.
- [21] Cristina Cifuentes. 1994. *Reverse compilation techniques*. Queensland University of Technology, Brisbane.
- [22] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *USENIX Sec.*
- [23] Al Danial. [n.d.]. CLOC. <https://goo.gl/3KFACB>.
- [24] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. 2020. Scalable Validation for Binary Lifters.
- [25] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPOSL*.
- [26] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *PLDI*.
- [27] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. BISTRO: Binary Component Extraction and Embedding for Software Security Applications.
- [28] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2018. rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery. In *ICCS*.
- [29] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *CC*.
- [30] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*.
- [31] Bauman Erick, Lin Zhiqiang, and Hamlen Kevin W. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS*.
- [32] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. 2018. Saluki: finding taint-style vulnerabilities with static property checking. In *NDSS*.
- [33] I. Guifanov. 2001. A Simple Type System for Program Reengineering. In *WCER*.
- [34] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.
- [35] Anastasis Keliris and Michail Maniatakos Yakdan. 2019. ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries. In *NDSS*.
- [36] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *ASE*.
- [37] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltamaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *ACSAC*.
- [38] Jakub Kroustek and Peter Matula. 2017. Retdec: An open-source machine-code decompiler. (2017).
- [39] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Sec.*
- [40] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*.
- [41] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*.
- [42] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*.
- [43] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *ASPOSL*.
- [44] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *ISSTA*.
- [45] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*.
- [46] Microsoft. 2018. llvm-mctoll. <https://github.com/Microsoft/llvm-mctoll>.
- [47] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *ICSE*.
- [48] Lily Hay Newman. 2019. The NSA makes Ghidra, a powerful cybersecurity tool, open source. <https://www.wired.com/story/nsa-ghidra-open-source-tool/>.
- [49] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *PLDI*.
- [50] National Security Agency (NSA). 2018. Ghidra. <https://www.nsa.gov/resources/everyone/ghidra/>.
- [51] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version Disassembly: Differential Testing of x86 Disassemblers. In *ISSTA*.
- [52] PNF. 2018. JEB Decompiler. <https://www.pnfsoftware.com/>.
- [53] PNF. 2018. Type Library. <https://www.pnfsoftware.com/blog/native-types-and-typelibs-with-jeb/>.
- [54] Thomas Reps and Gogul Balakrishnan. 2008. Improved Memory-Access Analysis for x86 Executables. In *CC*.
- [55] rev.ng Srls. 2018. Rev.ng. <https://rev.ng/>.
- [56] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables (*CCS '18*). Association for Computing Machinery, 426–441.
- [57] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *USENIX Sec.*
- [58] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *USENIX Sec.*
- [59] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*. Springer, 1–25.
- [60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*.
- [61] Dullien Thomas and Sebastian Porst. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*.
- [62] trailofbits. 2018. McSema. <https://github.com/trailofbits/mcsema>.
- [63] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, ZhaoFeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-scale Empirical Study on Mobile App Obsfuscation. In *ICSE*.
- [64] Ruoyu Wang, Yan Shoshtaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *NDSS*.
- [65] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *USENIX Sec.*
- [66] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 236–247.
- [67] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*.
- [68] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.



# Discovering Discrepancies in Numerical Libraries

Jackson Vanover

University of California, Davis  
United States of America  
jdvanover@ucdavis.edu

Xuan Deng

University of California, Davis  
United States of America  
xudeng@ucdavis.edu

Cindy Rubio-González

University of California, Davis  
United States of America  
crubio@ucdavis.edu

## ABSTRACT

Numerical libraries constitute the building blocks for software applications that perform numerical calculations. Thus, it is paramount that such libraries provide accurate and consistent results. To that end, this paper addresses the problem of finding discrepancies between synonymous functions in different numerical libraries as a means of identifying incorrect behavior. Our approach automatically finds such synonymous functions, synthesizes testing drivers, and executes differential tests to discover meaningful discrepancies across numerical libraries. We implement our approach in a tool named FPDIFF, and provide an evaluation on four popular numerical libraries: GNU Scientific Library (GSL), SciPy, mpmath, and jmat. FPDIFF finds a total of 126 equivalence classes with a 95.8% precision and 79.0% recall, and discovers 655 instances in which an input produces a set of disagreeing outputs between function synonyms, 150 of which we found to represent 125 unique bugs. We have reported all bugs to library maintainers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Software reliability; Empirical software validation.

## KEYWORDS

software testing, numerical libraries, floating point, differential testing

### ACM Reference Format:

Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering Discrepancies in Numerical Libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397380>

## 1 INTRODUCTION

Science and industry place their faith in numerical software to give accurate and consistent results. Numerical libraries make up the building blocks for such software, offering collections of *discrete numerical algorithms* that implement *continuous analytical mathematical functions*. In an effort to establish a trusted foundation from which to build powerful and useful tools, developers of numerical

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3397380>

libraries aim to offer a certain level of correctness and robustness in their algorithms. Specifically, a discrete numerical algorithm should not *diverge* from the continuous analytical function it implements for its given domain.

Extensive testing is necessary for any software that aims to be correct and robust; in all application domains, software testing is often complicated by a deficit of reliable test oracles and immense domains of possible inputs. Testing of numerical software in particular presents additional difficulties: there is a lack of standards for dealing with inevitable numerical errors, and the IEEE 754 Standard [1] for floating-point representations of real numbers inherently introduces imprecision. As a result, bugs are commonplace even in mature and widely-used pieces of numerical software [19], thus motivating the exploration of analysis and testing techniques.

With regard to the missing-oracle problem in the context of floating-point functions, a common technique involves taking the original function,  $F$ , allocating more bits per numerical variable to yield a "higher-precision" function  $F^*$ , and treating the output of  $F^*$  as the ground truth [9, 15, 16, 20, 42, 48, 49]. Such works rely on the assumption that  $F^*$  will have fewer *points of divergence* from the continuous analytical function, call it  $F_{\text{exact}}$ , and can thus be trusted as an approximation of  $F_{\text{exact}}$ . However, work by Wang et al. [44] regarding *precision-specific operations* illustrates the conditions under which this assumption fails, resulting in  $F^*$  potentially diverging even further from  $F_{\text{exact}}$ . Additionally, the execution of  $F^*$  can become prohibitively expensive: Benz et al. [9] report an increase in the time-to-complete that ranges from a factor of 167 $\times$  to 1016 $\times$  slowdown. Even in a situation in which  $F^*$  satisfies the above assumption, any points of divergence inherent to the algorithm itself will be present in both  $F$  and  $F^*$ . Comparing against the same discrete numerical algorithm, albeit at a higher precision, inherently limits the scope of discoverable bugs.

Consequently, this paper proposes a framework for automated and systematic differential testing as an orthogonal approach to complement the existing state-of-the-art. Our approach does not require a ground truth, and therefore, has a chance to uncover a larger variety of bugs. Rather than attempting to compare some numerical function  $F_1$  to  $F_{\text{exact}}$  or some approximation thereof, our approach compares  $n$  numerical functions,  $F_1, \dots, F_n$ , that all implement  $F_{\text{exact}}$ . We refer to such functions as *function synonyms*. Points of divergence found among these function synonyms and the discrepancies they cause are then reported.

While differential testing is a common technique that has been applied to many domains (e.g., [13, 14, 18, 31, 34, 40, 46]) such testing has not been systematically applied in the context of numerical software. In order to effectively implement differential testing in this domain and report meaningful points of divergence, we have identified three key challenges that must be addressed:

**Challenge 1: Automatically finding function synonyms.** Determining which functions implement the same continuous analytical mathematical function is non-trivial. Documentation can be unreliable and function names can also be misleading, e.g., the `sinc` function from `mpmath` is *not* a function synonym for the `sinc` functions from either `GSL` or `SciPy`, though `mpmath`'s `rf` is in fact a function synonym for `GSL`'s `poch`. Static approaches to discovering such synonyms in other domains [11, 17, 35, 36] suffer from a lack of parallel corpora or are based on calling context which, for numerical functions, can be widely-varied or even non-existent if being used for one-off calculations. This domain therefore calls for a dynamic approach.

**Challenge 2: Extracting signatures and synthesizing drivers.** Both finding function synonyms and testing them requires synthesizing drivers to evaluate such functions. Specifically, function signatures must be automatically extracted and each function must be wrapped in an executable driver. While the code for libraries written in statically-typed languages can be easily parsed to gather the information required for synthesizing such a driver, libraries written in dynamically-typed languages present unique difficulties in determining the types of function arguments. Again, documentation can be lacking and non-uniform; an alternative approach is required.

**Challenge 3: Identifying meaningful numerical discrepancies.** Because of the inherent imprecision of floating-point representations of real numbers, it is often the case that evaluating a pair of function synonyms over the same input will result in two outputs that, while not exactly equivalent, are not indicative of any buggy behavior. How do we know whether or not to call two outputs "equivalent"? Furthermore, the lack of overarching standards with respect to error-handling means that different libraries can produce different outputs when encountering the same erroneous computation. Such discrepancies often stem from deliberate developer choices and are not bugs. To better understand the reported points of divergence, a methodology to identify meaningful discrepancies that represent reportable bugs must be developed.

In this paper, we describe and implement FPDIFF, a tool for finding discrepancies between numerical libraries that addresses the above challenges. Given the source code of two or more libraries, FPDIFF starts by automatically extracting function signatures (Section 3.1.1) that fit our testing criteria (Section 2.3). To infer parameter types for functions from dynamically-typed libraries, FPDIFF leverages developer-written tests. Specifically, FPDIFF examines the Abstract Syntax Tree (AST) of test programs to infer argument types for each function call. These extracted function signatures are then used to automatically synthesize drivers to execute the functions (Section 3.1.2). When only *partial* type information is available, the space of possible parameter types is explored by generating multiple drivers, each with a unique configuration of data types.

FPDIFF then performs a classification step using the automatically-created drivers to evaluate each function over a set of *elementary inputs* that are designed to result in well-defined behavior (Section 3.2). Functions whose outputs consistently fall within the

same  $\mathcal{E}$ -neighborhood are determined to be function synonyms and placed in the same equivalence class.

This is followed by differential testing (Section 3.3) in which FPDIFF executes the drivers of function synonyms in each equivalence class on a diverse set of *adversarial inputs* designed to trigger divergence. Our trials include inputs that are manually crafted by developers, those that are found via binary guided random testing over a subset of the domain to find inputs that maximize inaccuracies in the output, and a set of inputs containing special values that are defined in the IEEE 754 Standard. The outputs across libraries are then compared to identify the points of divergence between function synonyms.

Finally, FPDIFF analyzes and reports numerical discrepancies (Section 3.3.2). Each of the resulting discrepancies is placed into one of six categories that are defined in Table 1. We developed this categorization based on our observations of the discrepancies generated by FPDIFF, their characteristics, and their likelihood of representing buggy behavior. Finally, a reduction of the set of discrepancies is automatically performed in order to facilitate manual inspection for bugs.

We perform an experimental evaluation over four numerical libraries: The GNU Scientific Library (`GSL`) [24] written in C, the JavaScript library `jmat` [43], and the Python libraries `mpmath` [26] and `SciPy` [27]. All of these libraries were chosen because they are open source, have overlapping functionality, and are widely used in practice. FPDIFF finds 655 unique discrepancies between these libraries and, of the reduced set of 327 discrepancies, we found 150 that represented 125 unique bugs. We have reported all bugs to library maintainers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

This paper makes the following contributions:

- We describe and implement FPDIFF, a tool for finding discrepancies between numerical libraries (Section 3).
- We propose a categorization of numerical discrepancies found across numerical libraries (Section 3.3.2).
- We present an evaluation on four popular numerical libraries: `GSL`, `mpmath`, `SciPy`, and `jmat` that finds a total of 655 unique discrepancies, 150 of which represent 125 bugs (Section 4).

The rest of this paper is organized as follows: Section 2 presents a motivating example, preliminary definitions, and describes the scope of our approach. Section 3 details the technical approach to each of the components that make up FPDIFF and Section 4 presents an experimental evaluation for each. We discuss related work in Section 5, and conclude in Section 6.

## 2 PRELIMINARIES

### 2.1 Illustrative Example

Reasoning about what is the ground truth when it comes to non-trivial floating-point computations is difficult. Short of deriving an analytical solution ourselves or consulting tables of analytically-derived results, true oracles are scarce. Checking against another function that claims to perform the same calculation is a possible strategy to investigate a suspicious result; such output comparisons illustrating unexpected or incorrect behavior are commonly found in bug reports related to numerical functions.

As a motivating example, consider *Tricomi's confluent hypergeometric function*, which provides a solution to Kummer's differential equation and has wide practical applications from pricing stock options [10] to calculating electron velocity in high-frequency gas discharge [32]. Using version 1.3.1 of the numerical library SciPy to compute the value of this function for the input values 0.0001, 1.0, and 0.0001, we see the following:

```
>>> from scipy import special
>>> special.hyperu(0.0001, 1.0, 0.0001)
-4806275004931.538
```

How can we determine the correctness of this result? Reading through some documentation and finding a function synonym of `hyperu` in version 1.0.0 of the `mpmath` library, we can attempt the same calculation, yielding a surprising discrepancy (albeit in `mpmath`'s own `mpf` data type):

```
>>> import mpmath
>>> mpmath.hyperu(0.0001, 1.0, 0.0001)
mpf('1.0009210608660066')
```

Another comparison against the implementation of the Tricomi function found in version 2.6 of GSL returns a result that agrees with `mpmath`. Furthermore, SciPy's documentation for this hypergeometric function provides no information regarding the domain of the inputs that they do or do not handle. All of this serves to indicate a likely bug in the SciPy library.

Library users often find such bugs by chance. A previous study [19] showed that indeed, the above methodology is a common practice among users of numerical libraries to confirm or rule-out spurious results. However, manual differential testing on a case-by-case basis is not only time consuming, but it also assumes that the user has already discovered a problematic input, and that identifying equivalent functions in other libraries (and writing tests for them) is straightforward. An automated framework to uncover numerical bugs in the first place rather than confirming suspicions after the fact is the motivation for the work presented in this paper.

## 2.2 Definitions

**Definition 2.1.** Given a set  $F$  of functions, we define a binary relation  $\triangleleft$  such that for any  $f_1, f_2 \in F$ , if  $f_1 \triangleleft f_2$ , then the set of analytical mathematical computations implemented by  $f_1$  are a subset of those implemented by  $f_2$ .

Consider the generic functions, `sqrt(x)` and `nthroot(n,x)`. We say that `sqrt`  $\triangleleft$  `nthroot`. It then follows that all of the functionality of `sqrt` is differentially testable against `nthroot`, but not vice versa.

**Definition 2.2.** Given a set  $F$  of functions, we define an equivalence relation  $\bowtie$  such that for any  $f_1, f_2 \in F$ , if  $f_1 \bowtie f_2$ , then  $f_1 \triangleleft f_2$  and  $f_2 \triangleleft f_1$ . We then say that  $f_1$  and  $f_2$  are **function synonyms**.

Note that the above definition of function synonyms does not describe instances in which two functions can be made "equivalent" via the fixing of certain parameters to a specific value, e.g., `sqrt(x)` and `nthroot(2,x)`, or the inlining of functions within larger expressions, e.g., `euclidean_norm(x,y)` and `sqrt(pow2(x)+pow2(y))`.

However, we do allow for a special case  $f_1 \bowtie f_2$  modulo data type in which the set of possible arguments given to either  $f_1$  or

$f_2$  is restricted to an infinite<sup>1</sup> subset of itself. For instance, though `gsl_bessel_In_scaled` from GSL and `ive` from SciPy both implement the exponentially-scaled modified Bessel function of the first kind, the former requires that the first parameter be an integer whereas the latter allows for a double-precision value. Therefore, `gsl_bessel_In_scaled`  $\bowtie$  `ive` but not vice versa. However, restricting the first parameter of `ive` to the integer data type results in `gsl_bessel_In_scaled`  $\bowtie$  `ive modulo data type`. We still refer to such cases as function synonyms.

**Definition 2.3.** Two function outputs  $y_1$  and  $y_2$  are considered  **$\mathcal{E}$ -equivalent** if they fall into one of three cases:

- 1)  $y_1, y_2 \in \mathbb{R}$  and  $|y_1 - y_2| < \mathcal{E}$ .
- 2)  $y_1, y_2 \in \{+\infty, -\infty, 0, \text{NaN}\}$  and are the same.
- 3)  $y_1$  and  $y_2$  are both exceptions.

**Definition 2.4.** A **discrepancy** is a tuple  $(C, p, \Phi, \mu)$  in which  $C$  is the equivalence class of function synonyms,  $p$  is the input that provoked the discrepancy (i.e., the point of divergence),  $\Phi$  is the set of outputs between which there exists at least one pair that is not  $\mathcal{E}$ -equivalent, and  $\mu$  is a unique hash value identifying the discrepancy (see Section 3.3.2).

## 2.3 Scope of Our Approach

In this work, we focus on discovering discrepancies between *special functions*, defined to be "function[s] (usually named after an early investigator of its properties) having a particular use in mathematical physics or some other branch of mathematics" [45]. Furthermore, we target special functions that are made publicly available via each library's API and are meant to be used standalone by a client program. This choice of scope naturally fits differential testing for several reasons: firstly, special functions are widely implemented by numerical libraries and there are many of them. For instance, special functions constitute between 16% to 84% of all API functions in `mpmath`, SciPy, GSL, and `jmat` with an average of 50%. Secondly, the algorithms used to implement them are complex and diverse with different developers making different choices on a case-by-case basis.<sup>2</sup> Lastly, the arguments and return values of special functions are elementary data types, thus facilitating input generation and output comparison. For all of these reasons, such a focus has precedent in the literature [7, 20, 21, 41, 42, 48, 49].

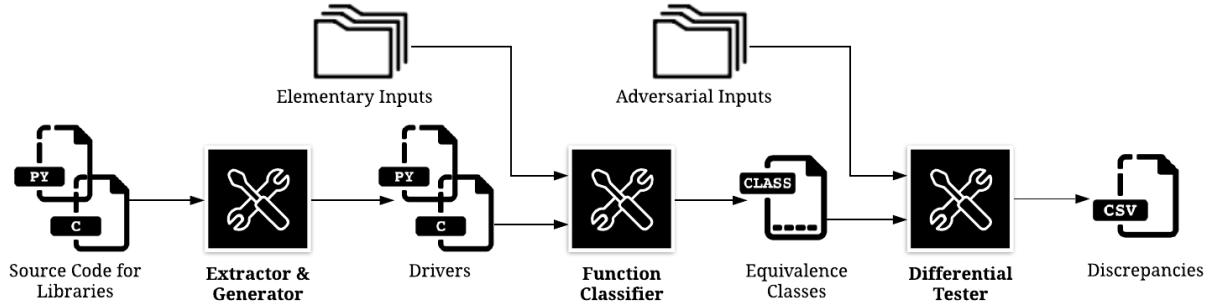
For any libraries that do not clearly delineate in the source code what is and is not a special function (e.g., via namespaces or directory structures), we approximate by targeting functions in which each parameter and return value represents a real number. Consequently, functions that work exclusively with complex numbers, matrices, and arrays are not targeted by the approach described in this paper. Such a specification is made in an effort to maintain a balance between function complexity and ease of comparability so that differential testing is fruitful.

## 3 TECHNICAL APPROACH

Figure 1 depicts the three main components of FPDIFF: (1) an extractor/generator that extracts signatures from library source code

<sup>1</sup>While the set of numbers representable with bits is not actually infinite, the term "infinite" here is used to refer to the set of numbers *represented* in memory.

<sup>2</sup>Interested readers are referred to *Numerical Methods for Special Functions* [23].



**Figure 1: An overview of our framework which automatically collects function signatures, synthesizes executable drivers, discovers function synonyms, and applies differential testing to find points of divergence between numerical libraries.**

and synthesizes executable drivers for numerical functions, (2) a classifier that clusters functions into equivalence classes of function synonyms based on their outputs when evaluated on *elementary inputs*, and (3) a differential tester that executes drivers within each equivalence class on a set of *adversarial inputs* and reports discrepancies found between function synonyms.

### 3.1 Drivers for Numerical Functions

**3.1.1 Signature Extraction.** Given a library’s source code, the extractor returns a list of function signatures and the names of any potential imports/header files that might be required to call the functions. The strategy for extracting this information is dependent on each language’s handling of types and, furthermore, on each library’s conventions for declaring functions. In this paper, we implement FPDiff to target libraries written in both statically- and dynamically-typed languages.

**Signatures when given type information.** The inclusion of type information allows us to tailor the extractor to target only those functions that fit our scope, i.e., those with at least one double value in the input and that return a double value as the output. Once the library convention for defining functions is identified, a simple parsing of the source code for matching patterns suffices.

**Signatures when given no type information.** A potential source of type information is parsing the library’s documentation, but the quality and consistency of that documentation becomes the limiting factor. For instance, mpmath makes no mention in the text of its documentation about the types of function parameters. Instead the user is expected to reason about types based on typeset mathematical formulas included in the documentation.

To address this challenge, we leverage developer-written tests to gather partial type information. To this end, we parse and traverse the AST for each test file to collect functions that, when called, have arguments that are either literals of type `int` or `double` or whose arguments are variables containing a reference to such values. In lieu of complete type information, all of these numerical arguments are collected and associated with the name of the called function. Additionally, we collect import statements found in these tests.

While this approach helps to filter out functions that do not take numerical arguments, it is limited: the literal arguments used to test

the functions do not reliably indicate the type of the corresponding function parameter, e.g., when passing an integer 1 as an argument that the corresponding function synonym in a statically-typed language might require to be a double 1.0. Therefore, while the former extraction technique creates a collection of definitive function signatures with the desired parameter and return types, this technique yields a collection of function signatures that, when called in the library tests, had only numerical arguments. Such functions may have return values that fall outside of our scope. The technique for overcoming this ambiguity is addressed via instrumentation added to the drivers during their generation (detailed below) and the classification algorithm shown in Algorithm 1 and described in Section 3.2.

**3.1.2 Driver Generation.** Because FPDiff executes each function to identify function synonyms (Section 3.2) and to discover points of divergence (Section 3.3.2), the goal of the driver generator is to automatically create a harness around each function that facilitates the function evaluation.

The driver generator receives input from the extractor consisting of a collection of function signatures (with or without complete type information) and any necessary header files or imports. The generator inserts this information into a template to create drivers like the GSL and SciPy drivers shown in Figure 2. Each driver must meet the following criteria:

- (1) Feed arguments from a uniform set of inputs to facilitate both the placement of the functions into equivalence classes based on their evaluation over elementary inputs and the subsequent differential testing over the set of adversarial inputs. This is accomplished by parameterizing each driver with two arrays of double-precision floats and integers (first lines of Figures 2a to 2c) such that a function with  $n$  parameters will involve  $n$  inputs read from the appropriate arrays (highlighted lines in Figures 2a to 2c).
- (2) Allow for exceptions within the library code to propagate upward to the calling procedure to be collected. For example, Line 4 in Figure 2a accomplishes this by overriding the default error handler for GSL, which aborts program execution.
- (3) Provide a means of raising an appropriate exception when attempting to execute a driver for a function which, though found

```

1 double gsl_sf_bessel_In_scaled_DRIVER(char*
2     doubleInput, char* intInput) {
3
4     double out;
5
6     gsl_error_handler_t * old_handler =
7         gsl_set_error_handler (&my_handler);
8
9     out = gsl_sf_bessel_In_scaled(intInput[0],
10        doubleInput [0]);
11    return out;
12}

```

(a) Definitive driver synthesized for the GSL function `gsl_sf_bessel_In_scaled`.

```

1 def scipy_special_ive_DRIVER0(doubleInput, intInput):
2     out = special.ive(doubleInput[0], doubleInput[1])
3
4     return float(out)

```

(b) One of the two drivers synthesized for the SciPy function `ive`. Though executable, this driver will not be mapped to the analogous GSL function shown in (a) because `ive` will not be fed the same inputs.

```

1 def scipy_special_ive_DRIVER1(doubleInput, intInput):
2     out = special.ive(intInput[0], doubleInput[0])
3
4     return float(out)

```

(c) Second driver for the SciPy function `ive`. This driver will successfully be mapped to the analogous GSL function shown in (a).

**Figure 2: Examples of Generated Drivers**

by the signature extractor, does not fit our chosen scope, e.g., a function that was present in the developer tests with double inputs but returns arrays. Line 3 in Figures 2b and 2c accomplishes this by raising an exception when the return of the function call is not a floating-point number.

- (4) For those functions for which we do not have complete type information, the driver must utilize the proper configuration of integer and double arguments to be correctly mapped to potential function synonyms. See discussion below.

Fulfillment of the fourth criteria allows our pipeline to automatically perform differential testing between libraries that do not provide type information. Consider the example from Section 2.2 `gsl_sf_bessel_In_scaled` ↔ `ive` mod data type.

Because the signature for `gsl_sf_bessel_In_scaled` was extracted from a library with type information, the generator synthesizes the single definitive driver shown in Figure 2a in which the first parameter is an integer and the second is a double. However, even though `ive` from SciPy is the proper function synonym, its signature was extracted without type information. Thus, `ive` will only appear functionally identical to `gsl_sf_bessel_In_scaled` if given the same argument types. In other words, we have to ensure that the first argument to `ive` comes from the array of integer inputs just like `gsl_sf_bessel_In_scaled`.

In order to address this challenge, the driver generator will create multiple drivers for each extracted function signature that lacks complete type information, each with a different configuration of integer and double data types. Figures 2b and 2c illustrate the two

drivers synthesized for the SciPy function `ive` with their difference highlighted. During the classification process discussed in Section 3.2, only the drivers with matching data type configurations (in this case, Figure 2a and Figure 2c) will be recognized by the classifier as function synonyms.

### 3.2 Function Classification

Our classifier addresses one of the main challenges of differential testing: identifying what function pairs are "equivalent" and therefore acceptable for such testing. A set of drivers is supplied as input to the classifier which outputs a set of equivalence classes that indicate the existence of likely function synonyms (Definition 2.2).

Algorithm 1 describes our methodology. First, each function is assigned a *characteristic vector* of numerical values that aims to represent the underlying semantics of the function (Lines 3 through 8). Equivalence classes are then constructed based on the similarity of these vectors (Lines 11 through 19). To generate each characteristic vector, we leverage a property of any relatively well-written numerical algorithm: for the overwhelming majority of its input domain, the function is well-behaved [37]. Note that because the space of possible floating-point values is so vast, each output inherently encodes detailed information about the underlying calculation used to arrive at that value: a collection of such outputs therefore captures a portion of the semantics of the program that generated it in a form that is easily comparable via standard arithmetic operations. In contrast, a numerical function that returns integer values has outputs that contain much less information about the program's underlying semantics by way of the pigeonhole principle.<sup>3</sup>

To this end, our classifier evaluates each synthesized driver over a set of *elementary inputs*. Elementary inputs are values for which most functions are likely to exhibit well-defined behavior; the resulting set of outputs constitutes the characteristic vector. Note that the else branch on Line 7 of Algorithm 1 handles the possibility that an evaluation over an elementary input does not exhibit well-defined behavior by logging an output of NaN. Furthermore, the conditional statement on Line 9 removes unwanted drivers that exhibited undesirable behavior on every evaluation over every elementary input and, therefore, have a characteristic vector of all NaN values; this works in tandem with the driver instrumentation required by the third criteria for successful drivers as stated in Section 3.1.2.

For the classification based on characteristic vectors, the conditional on Line 13 ensures that only functions with the same number and types of parameters are compared. Finally, the conditional on Line 14 implies an element-wise comparison of the elements in each characteristic vectors for  $\mathcal{E}$ -equivalence (see Definition 2.3). Note that the size of the  $\mathcal{E}$ -neighborhood varies depending on the size of the values being compared; intuitively, larger values will have larger  $\mathcal{E}$ 's. To address this, FPDIF users define a relative tolerance  $\epsilon$  such that if the relative error between any two values is less than  $\epsilon$ , the values will be considered  $\mathcal{E}$ -equivalent.

### 3.3 Differential Testing

Our differential tester takes a set of equivalence classes and evaluates drivers over a set of *adversarial inputs*, i.e., inputs that are

<sup>3</sup>[https://en.wikipedia.org/wiki/Pigeonhole\\_principle](https://en.wikipedia.org/wiki/Pigeonhole_principle)

**Algorithm 1:** Function Classification

---

```

Input: Drivers, elementaryInputs
Output: equivalenceClasses
1 equivalenceClasses = []
2 for driver in Drivers do
3   for doubleInput, intInput in elementaryInputs do
4     result = driver(doubleInput, intInput)
5     if isNumerical(result) then
6       driver.characteristicVec.append(result)
7     else
8       driver.characteristicVec.append(NaN)
9     if allNaN(driver.characteristicVec) then
10      continue
11    match = False
12    for class in equivalenceClasses do
13      if sameParameterConfig(class[0], driver) then
14        if sameCharacteristicVec(class[0], driver) then
15          match = True
16          class.append(driver)
17          break
18      if not match then
19        equivalenceClasses.append([driver])
20 return equivalenceClasses

```

---

intended to provoke divergences between function synonyms. Outputs are then compared for  $\mathcal{E}$ -equivalence with points of divergence reported as a *discrepancy* (see Definitions 2.3 and 2.4).

There are three remaining challenges to enable differential testing of numerical functions: (1) How do we obtain effective adversarial inputs? (2) How do we triage discrepancies found? (3) How do we automatically reduce the set of reported discrepancies to facilitate manual inspection? We discuss these challenges below.

**3.3.1 Sources for Adversarial Inputs.** The set of inputs for which a function produces significant divergences is infinitesimal compared to the size of the possible domain [48, 49]. It is this fact that both motivates our technique for classifying functions (Section 3.2) and represents one of the major challenges for differential testing of numerical libraries. Discovering such inputs is a well-studied problem [5, 9, 15, 16, 21, 25, 49]; though we choose three sources of inputs in this paper, it is worth noting that other input sources may be used. For this work, our adversarial inputs come from tests written by developers, a guided binary-search of a portion of the domain, and special values as defined by the IEEE 754 Standard.

**Test Migration Inputs.** With test migration, we leverage the assumption that inputs used in test programs are carefully chosen by developers to expose corner cases in the algorithms being tested. During the signature extraction process described in Section 3.1.1, we collect such inputs and associate them with the function signature they were used with in order to migrate them to function synonyms within the same equivalence class.

**Inaccuracy-Inducing Inputs.** For each pair of function synonyms and for a specified domain range, we perform a binary guided random search for inputs that maximize the relative error between the two functions.<sup>4</sup> These inputs are then collected and applied to the rest of the functions in the equivalence class. The assumption here is that inputs found to cause inaccuracies between one pair of function synonyms will have a higher likelihood of prompting divergence between other functions within the equivalence class.

**Special-Value Inputs.** In addition to the finite set of floating-point numbers representable within a particular format, the IEEE 754 Standard also defines a set of *special values*: negative zero, positive infinity, negative infinity, and NaN.<sup>5</sup> These values can arise as the result of certain floating-point operations and, as such, give rise to the possibility that they might be used as input to another numerical function; therefore, while not strictly a part of an algorithm to implement a mathematical function, it is still the responsibility of developers to handle such inputs. Moreover, documentation for libraries is often lacking when it comes to describing behavior for inputs outside the expected domain if it mentions domain at all. Using these values as adversarial inputs in our differential testing will illustrate the choice (or lack thereof) that different developers make on how to handle special values.

**3.3.2 Discrepancy Triage.** Recall that each discrepancy is defined as a tuple  $(C, p, \Phi, \mu)$  representing respectively the equivalence class of function synonyms, the point of divergence, the outputs exhibiting the divergence, and a unique identifying hash value (see Definition 2.4).

Our analysis first maps each element of  $p$  to an element of the abstract domain consisting of NaN, infinite, zero, negative, and positive. The result is the abstract input  $\hat{p}$ . For example, if  $p = \{0, -1.1, -\inf\}$ , then  $\hat{p} = \{\text{zero}, \text{negative}, \text{infinite}\}$ . Each  $\hat{p}$  is then used to calculate  $\mu$ , described in Section 3.3.3, and categorize the discrepancy.

The discrepancy categories are summarized in Table 1 and are loosely ordered by their likelihood of representing buggy behavior. Such a categorization is intended to group together discrepancies that likely indicate similar undesirable behaviors in one or more libraries and serves as one of our key contributions that enable FPDIFF to effectively report bugs. These categories are a result of a manual inspection of discrepancies discovered via our testing framework. Each category is described and discussed below.

*Category 6 discrepancies are those in which a hang is observed in at least one of the libraries.* Such discrepancies clearly represent buggy behavior; programs should terminate gracefully regardless of input, making a hang unacceptable.

*Category 5 discrepancies are those in which a double output is generated in at least one of the libraries from a  $\hat{p}$  containing a NaN value.* A NaN input could be the return value of another function indicating that some illegal operation took place (e.g., zero divided

<sup>4</sup>Chiang et al. [16] find inputs that maximize error between a program and its high-precision version, and focus on C programs. We adapt their technique to maximize error across distinct implementations of numerical functions, written in Python and C.

<sup>5</sup>Though the IEEE 754 Standard defines a quiet NaN and a signalling NaN, the languages we focus on do not define the behavior of the latter and use NaN to refer to the former.

**Table 1: A categorization of discrepancies, loosely ordered by likelihood of representing buggy behavior.**

Category	Discrepancy Description
6	A hang is observed in at least one of the libraries.
5	Double output is generated in at least one of the libraries from a $\hat{p}$ containing a NaN value.
4	Mix of doubles, exceptions, and/or special values from a $\hat{p}$ containing an infinite value.
3	A relative error greater than a chosen $\epsilon$ is detected between two libraries.
2	Mix of doubles, exceptions, and/or special values generated from a $\hat{p}$ containing only positive, negative, or zero values.
1	Mix of special values and exceptions generated from any input type.

by zero, logarithm of a negative number, etc.). If the function receiving that NaN value produces a double value as result, then the fact that an illegal operation occurred prior to the function call is lost. In such cases, it is also troubling to consider the fact that an input representing "not-a-number" somehow gave rise to a number as an output. Category 5 discrepancies have a high potential for representing true bugs.

*Category 4 discrepancies are those in which a mix of doubles, exceptions, and/or special values are generated from a  $\hat{p}$  containing an infinite value.* The existence of such discrepancies may be due to differing design choices between libraries on whether or not to support the evaluation of limits approaching infinity. Domain-specific knowledge regarding the asymptotic behaviors of the special function in question is required to determine whether or not each discrepancy represents a true bug.

*Category 3 discrepancies are those in which a relative error greater than a chosen  $\epsilon$  is detected between two libraries.* This indicates that the libraries cannot agree on the result of a legitimate computation; the seriousness of such an error has been well-documented with tragedies such as the Patriot Missile failure and the explosion of the Ariane 5 rocket [3, 4].

*Category 2 discrepancies are those in which a mix of doubles, exceptions, and/or special values are generated from a  $\hat{p}$  containing only positive, negative, or zero values.* These discrepancies suggest that the libraries do not agree on whether or not a problem occurred in the computation. Such ambiguity is obviously undesirable, though diagnosing this behavior as buggy can require substantially more manual effort than the other categories above; see Section 4.3 for a discussion of some specific examples.

*Category 1 discrepancies are those in which a mix of special values and exceptions are generated from any input type.* These discrepancies are distinct from those in categories 2, 4, and 5 because no double outputs are observed. This indicates that, while the libraries all appear to agree on the fact that a problem occurred in the computation, they disagree on how to handle such a problem. Category 1 discrepancies illustrate the lack of unifying standards for error handling numerical libraries, and do not necessarily constitute reportable bugs.

While some of these categories are potentially discoverable by systematically executing drivers over different inputs, doing so within equivalence classes of function synonyms gives users the added benefit of either suggesting the correct behavior (when there

is a general consensus between the other function synonyms) or demonstrating the variety of different choices made by developers in handling adversarial inputs.

As an additional measure to aid in discrepancy triage, FPDIFF also utilizes a  $\delta$ -difference metric as described by Klinger et al. [29] that quantifies for each function how many other function synonyms within the equivalence class have outputs that disagree with that function. For example, if function  $f_1$  returns NaN and its function synonyms  $f_2$ ,  $f_3$ , and  $f_4$  all return 0 for a specific adversarial input, the  $\delta$ -differences will be 3 for  $f_1$  and 1 for the others. FPDIFF tags each discrepancy with the maximum  $\delta$ -difference between all the function synonyms. Larger values will therefore correspond to discrepancies present in larger equivalence classes in which a minority of function synonyms disagree with the majority.

**3.3.3 Reducing Reported Discrepancies.** In order to minimize the reporting of multiple discrepancies that are all representative of the same buggy behavior, we ensure that the each discrepancy output by the differential tester is assigned a  $\mu$  value that encodes the defining characteristics of each discrepancy: the equivalence class of function synonyms in which the discrepancy was observed, the point of divergence, and the set of results that disagree. With this in mind,  $\mu$  is calculated as the hash value of the input tuple consisting of the summation of the equivalence class' characteristic array, the abstracted input tuple  $\hat{p}$ , and the category of the discrepancy. In the case of multiple category 3 discrepancies with the same  $\mu$ , the one with the largest relative error is reported. In all other cases, the first discrepancy found is the one reported.

Additionally, all category 1 discrepancies are removed from the final reduced log of discrepancies. As discussed above, such discrepancies indicate the different techniques library developers use to handle inevitable numerical errors. While their existence is informative in its own right, they do not represent reportable bugs.

## 4 EXPERIMENTAL EVALUATION

This evaluation is designed to answer the following questions:

- RQ1** How effective are our signature extractor and driver generator at discovering functions and synthesizing their drivers?
- RQ2** How effective is our classifier at discovering function synonyms within and across numerical libraries?
- RQ3** What sorts of discrepancies does FPDIFF discover between function synonyms?
- RQ4** Does differential testing discover discrepancies that represent reportable bugs? If so, how difficult is it to ascertain which discrepancies are reportable?

*Selection of Numerical Libraries.* We consider four numerical libraries: the GNU Scientific Library (GSL), SciPy, mpmath, and jmat. We chose open-source numerical libraries that have overlapping functionality, are widely-used in practice, and cover a set of very different programming languages. GSL is a C library of numerical routines whose special functions are a popular subject of experimental evaluation for numerical testing like this (e.g., [7, 21, 22, 41, 42, 48, 49]). SciPy is a well-known Python numerical library that implements a variety of numerical routines and is used by more than 136,000 GitHub repositories. mpmath is a Python numerical library that implements much of the functionality of SciPy with additional support for arbitrary-precision arithmetic and is a standard package included in both the SymPy and Sage computer algebra systems. jmat is a JavaScript numerical library that provides implementation for special functions, linear algebra, and more; it is the most popular JavaScript library with this functionality found on GitHub.

*Experimental Setup.* The classifier uses a relative tolerance value of  $\epsilon = 10^{-8}$  (on the order of machine epsilon) to determine  $\mathcal{E}$ -equivalence. FPDIFF uses a relative tolerance value of  $\epsilon = 10^{-3}$  to determine what amount of inaccuracy is considered severe enough to report as a category 3 discrepancy (previously, we used the same relative tolerance as the classifier but initial feedback showed that some developers considered such inaccuracies too small to be significant, even if their library was the only outlier). In the absence of a ground truth by which to scale the absolute error, we use the magnitude of the relative percent difference, shown in equation 1, as our relative error. Any references to relative error within our evaluation are taken to mean the relative percent difference.

$$\frac{|x_1 - x_2|}{(|x_1| + |x_2|)/2} = 2 \frac{|x_1 - x_2|}{|x_1| + |x_2|} \quad (1)$$

The classifier executes each driver over 40 elementary inputs to yield characteristic vectors. Elementary inputs of double type are randomly and uniformly generated between 0 and 3 exclusive and integer inputs are either 0, 1, 2, or 3. The choice of elementary inputs was guided by intuition and refined heuristically. Roughly speaking, most special functions tend to exhibit their most interesting/defining behavior around zero and are more likely to be defined for positive values than negative values.

Hangs constituting category 6 discrepancies are discovered by setting a timeout threshold for each function’s execution. For our experiments, we chose a time limit of 20 seconds.

We generated a collection of inaccuracy-inducing inputs using a modified version of the s3fp tool developed by Chiang et al. [15] which uses binary guided random testing to find inputs that maximize the floating-point error between 128-bit precision and 64-bit precision versions of a single C program. We modified their source code to also work with Python programs, to use 64-bit double data types, and to calculate relative errors with equation 1. For each pair of function synonyms, we allowed an hour of search time within the domain interval  $[-100, 100]$  as done in their original evaluation.

Each function in the mpmath library is present in two different namespaces: one that corresponds to an arbitrary-precision implementation and one for a floating-point-precision implementation. Using this knowledge, the driver generator gives us a pair of drivers

**Table 2: Evaluation of the Driver Generator**

Library	Eligible Functions	# Captured	% Captured
GSL	193	193	100%
jmat	154	154	100%
mpmath	211	147	69.7%
SciPy	206	180	87.4%
Total	764	674	88.2%

for each mpmath function. In the following evaluation, we do not count these pairs as function synonyms discovered by FPDIFF, we do not double count mappings to each mpmath function, and we do not count equivalence classes that only contain these pairs.

FPDIFF is implemented as a collection of Python modules which can be found at <https://github.com/ucd-plse/FPDiff>. The documentation includes directions for those who wish to extend FPDIFF to add new libraries for testing. We ran our experiments on a workstation with a 3.60GHz Intel i7-4790 and 32 GB of RAM running Ubuntu 18.04.

## 4.1 Effectiveness of Extractor and Driver Generator

To evaluate the effectiveness of the signature extractor and driver generator, we report the recall with respect to drivers synthesized by FPDIFF for functions eligible for our experimental evaluation. We omit precision calculations because, as discussed in Section 3.1.2, the extractor/generator component creates drivers for a superset of the targeted functions (in the case of libraries without type information) which is handled with a combination of driver instrumentation and the function classification described by Algorithm 1. Also note that these components work in tandem and, as such, are evaluated together; if the extractor successfully captured a function signature, the generator will synthesize a driver for that signature.

We manually examine the documentation for the chosen libraries to construct the ground truth of eligible functions for our study (see Section 2.3 for the scope). For SciPy and mpmath’s functions that include optional parameters with default arguments, we count each possible configuration as a unique function signature. The results are shown in Table 2.

Overall, we successfully generate an executable driver for 674 out of 764 eligible functions (88.2%). Note that the 100% recall for GSL and jmat is unsurprising due to the fact that both libraries include type information in their signatures, allowing us to simply parse source code for matching patterns.

We found the success of these components to be highly correlated with the thoroughness of library test code. For example, of the 64 mpmath functions we did not capture, 57 of them were not present at all in the developer tests. The remaining 7 functions were not captured because they were tested with named constants defined elsewhere in the library or arguments to lambda expressions; in these cases, our extractor was not able to determine that these function’s parameters were numerical arguments as required by our experiment’s scope.

**Table 3: Function Synonym Mappings Precision and Recall**

Library Pair	Precision	Recall
GSL/jmat	27/27 (100%)	27/38 (71.1%)
GSL/mpmath	61/65 (93.8%)	57/69 (82.6%)
GSL/SciPy	68/69 (98.6%)	67/76 (88.2%)
jmat/SciPy	29/30 (96.7%)	27/43 (62.8%)
mpmath/jmat	44/45 (97.8%)	40/49 (81.6%)
mpmath/SciPy	47/48 (97.9%)	44/56 (78.6%)
same library	21/26 (80.8%)	16/21 (76.2%)
Total	297/310 (95.8%)	278/352 (79.0%)

While all but two of the eligible SciPy functions appeared in the developer written tests, 24 of the 26 functions that did not receive executable drivers were missed by the extractor because they were either passed as function objects without parameters to a separate testing function or their arguments were `numpy.ndarray` objects.

**RQ1:** It is not difficult to obtain a 100% capture rate for libraries containing type information; for those without such information, the success of leveraging developer tests is highly dependent on the quality of those tests. More mature libraries such as SciPy are more likely to have more exhaustive tests which benefits our technique.

## 4.2 Effectiveness of Function Classifier

To establish the ground truth for the set of mappings FPDIFF should capture, we manually placed all of the eligible functions into equivalence classes based on documentation, source code, and other external resources. Our function classifier reported 310 pairs of function synonyms with 95.8% precision and 79.0% recall. A breakdown per-library is shown in Table 3. The set of correct function synonyms constituted 126 equivalence classes, encompassing 498 functions ready for differential testing.

Precision was calculated by manually verifying each mapping reported by FPDIFF. The source of imprecision in all cases was the choice of elementary inputs; while equivalent for positive inputs, these false function synonyms diverge when the inputs are negative. It is worth noting that while there were 13 incorrect mappings, these were the consequence of only 7 misplaced functions.

To demonstrate the effectiveness of our classifier technique, consider the set of Bessel functions. Between the four libraries, there are about 90 such functions with each library using their own naming conventions to distinguish between them. FPDIFF discovers 28 equivalence classes of Bessel functions, one of which encompasses the regular cylindrical Bessel functions of the first kind with integer order, shown below:

$$\begin{cases} \text{gsl_sf_bessel\_Jn} & (\text{GSL}) \\ \text{angerj} & (\text{jmat}) \\ \text{besselj} & (\text{mpmath}) \\ \text{jn} & (\text{SciPy}) \\ \text{jv} & (\text{SciPy}) \\ \text{jve} & (\text{SciPy}) \end{cases}$$

First, such mappings are non-obvious based on names alone. For instance, `gsl_sf_bessel_jl` and `gsl_sf_bessel_Jnu` are correctly determined to *not* belong to this equivalence class. Second, we see a valid intra-library function synonym, `jn`  $\Leftrightarrow$  `jv`, despite the fact that `jn` does not exist in the SciPy documentation. Third, we observe a number of valid function synonyms *mod data type* (see Section 2.2): `gsl_sf_bessel_Jn` is the implementation of the regular cylindrical Bessel function of the first kind with *integer* order. `jn`, `jv`, and `besselj` implement the same function, but for any *fractional* order. `jve` is the same as `jn` and `jv`, but scaled by an exponential of the imaginary part of the input. `angerj` is a generalization of the Bessel function of the first kind. All of these become synonyms *mod data type* when their first parameter is an integer.

As for recall, the main causes for missing mappings were a failure in the extractor to get the function in the first place or inaccuracies in a function's evaluation over elementary inputs that exceeded the choice of  $\epsilon$ . For instance, while FPDIFF reported the six functions above for the equivalence class of cylindrical Bessel functions of the first kind with integer order, it failed to discover a seventh, the `jmat` function `besselj`. This is because inaccuracies in the function caused several of the elements in its characteristic vector to fall outside of the acceptable  $\mathcal{E}$ -neighborhood. However, FPDIFF did correctly place `jmat`'s `besselj` in the equivalence class representing the regular cylindrical Bessel functions of the first kind with *fractional* order mentioned above.

Note that in the set of reported equivalence classes, there existed 5 correctly mapped functions that were not in the set of eligible functions that we manually constructed. This was because we used the documentation as reference when constructing the ground truth and these 5 functions were not listed in the documentation (as in the case of `jn` above). Any mappings to these functions were therefore not counted in the recall portion since they were not in the ground truth set.

**RQ2:** FPDIFF finds function synonyms with a 95.8% precision and 79.0% recall, thus demonstrating the effectiveness of classifying functions based on their evaluation over elementary inputs. This approach even managed to correctly place functions that were not present in the documentation. However, efficacy is influenced by the choice of elementary inputs. Improvements in the extractor/generator component would benefit recall.

## 4.3 Discrepancies Found

After removing discrepancies with identical values of  $\mu$ , FPDIFF found a grand total of 655 unique discrepancies between function synonyms from the GSL, `jmat`, `mpmath`, and SciPy numerical libraries. For the manual inspection for bugs that followed, all 328 category 1 discrepancies were removed to give a reduced set of 327. Of these, we found 150 of them to represent 125 unique bugs. Specifically, FPDIFF identified 31 bugs in GSL, 33 bugs in `jmat`, 44 bugs in `mpmath`, and 17 bugs in SciPy. We have reported all bugs to library developers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

Table 4 shows the counts of unique discrepancies found by FPDIFF broken down by adversarial input source and discrepancy

**Table 4: Discrepancies found per input source and category.**

	Category #						Total
	6	5	4	3	2	1	
Special Values	20	23	95	7	11	280	436
Test Migration	-	-	-	43	107	48	198
Inaccuracy-Inducing	-	-	-	13	12	1	26
Overall (Unique)	20	23	94	61	129	328	655
<b>Reportable</b>	<b>19</b>	<b>21</b>	<b>37</b>	<b>23</b>	<b>50</b>	<b>0</b>	<b>150</b>

**Table 5: Bugs found per library and discrepancy category.**

	Category #						Total
	6	5	4	3	2	1	
GSL	-	8	12	2	9	-	31
jmat	-	5	8	8	12	-	33
mpmath	17	7	10	1	9	-	44
SciPy	-	7	2	2	6	-	17
Total	17	27	32	13	36	-	125

category (see Table 1 for category descriptions), as well as counts of discrepancies containing reportable buggy behavior and unique bugs found per discrepancy category. In total, FPDIFF used 160 special-value inputs, 4,513 migrated test inputs, and 1,092 inaccuracy inducing inputs. Each test-migration and inaccuracy-inducing input was applied only to the equivalence class from which it was found while each of the special value inputs was applied to *all* equivalence classes. This combined with the fact that special value inputs were the only adversarial inputs to yield category 4, 5, and 6 discrepancies explains why the relatively smaller number of unique inputs generated so many more discrepancies. Migrated test inputs were significantly more effective than inputs discovered via a binary-guided random search, generating over 7 times as many overall discrepancies and over 3 times as many inaccuracies; this lends support to our hypothesis that inputs used in test programs are carefully chosen by developers to expose corner cases in the algorithms being tested.

Table 5 shows the break down of reportable bugs per library and discrepancy category. FPDIFF helped us to identify the most bugs in mpmath, followed by jmat, GSL, and SciPy. With the exception of category 6, the distribution of bugs across libraries was mostly even (with a couple of outliers in categories 3 and 4, see discussion below). This demonstrates that the discrepancies found by FPDIFF and the bugs that they represent are not specific to a single numerical library. Note that categories 2 and 4 represented the highest number of bugs (36 and 32, respectively) while category 3 represented the fewest (13 bugs).

We now offer a series of case-studies surrounding examples discovered by FPDIFF from each discrepancy category.

**4.3.1 Category 6 Discrepancies.** Because hangs are certainly buggy behavior, every category 6 discrepancy has a high probability of representing a reportable bug. Points of divergence that caused hangs all included NaN or infinite values, leading to a natural suggestion

for a fix: checking inputs for special values. Such hangs were only observed in mpmath with only 1 of the 20 not being reproducible (this instance was due to an exception being raised regarding lack of convergence when the function's execution was allowed to exceed our chosen cutoff). While these hangs were acknowledged by mpmath developers, fixes have yet to be applied, which might indicate that the apparent simplicity of the fix may be misleading (see discussion below).

**4.3.2 Category 5 Discrepancies.** Consider the following discrepancy discovered by FPDIFF:

```
(jmat) factorial(NaN) => NaN
(mpmpath) mp.factorial(NaN) => NaN
(mpmpath) fp.factorial(NaN) => EXCEPTION
(SciPy) factorial(NaN) => 0
```

The majority behavior of propagating the NaN, as observed in jmat and the arbitrary-precision mpmpath implementations, suggests a more appropriate action than simply returning 0. After filing a report, this was confirmed in the ensuing discussion with SciPy developers, who stated, "inputs in the array that enter as NaN should remain NaN." Thus, our bug report ultimately led to a fix to be included in the 1.5.0 release of SciPy.

Note that even though such fixes may be conceptually simple (check input for NaN, return a NaN if found, otherwise return the function result), the complex structure of numerical software makes for a non-trivial implementation. For this particular example with `scipy.special.factorial`, the pull request (PR) containing only this bug fix included 8 commits, 26 changes, and a back-and-forth between maintainers encompassing dozens of comments that lasted almost a month. Interested readers are encouraged to examine the thread accompanying the PR.<sup>6</sup>

It is also worth noting the discrepancy between the arbitrary-precision and floating-point-precision functions in mpmpath. The floating-point implementation throws an exception with the message "cannot convert float NaN to an integer" while the arbitrary-precision implementation propagates the NaN value. It is not difficult to contrive a scenario in which a user swaps the two implementations within some larger program, assuming they are equivalent, thus possibly giving rise to unexpected failures.

**4.3.3 Category 4 Discrepancies.** Consider the following discrepancy discovered by FPDIFF between functions implementing the *Dirichlet eta function*, also known as the *alternating zeta function*:

```
(GSL) gsl_sf_eta(inf) => 1
(jmat) eta(inf) => NaN
(mpmpath) mp.altzeta(inf) => 1
(mpmpath) fp.altzeta(inf) => 1
```

Again, the majority-rules intuition suggests that this function asymptotically approaches 1. A confirmation of this via an external resource and the fact that the jmat library supports evaluation of limits led to a bug report and a fix.

Note from Table 5 that only 2 out of 32 bugs related to category 4 discrepancies were attributed to SciPy. This might indicate

<sup>6</sup><https://github.com/scipy/scipy/pull/11254>

that SciPy has a more consistent policy for the evaluation of limits in special functions than the other libraries.

**4.3.4 Category 3 Discrepancies.** The example shown in Section 2.1 is a category 3 discrepancy that was discovered by FPDIFF using a migrated test input from GSL. Searching the issue tracking system for SciPy showed similar reports dating back to 2013 regarding such inaccuracies in the *Tricomi confluent hypergeometric function*. A recent comment from a developer on an issue filed in late 2019 stated, "hyperu definitely needs more improvements. We're getting there, but it's a process." This demonstrates the fact that, unlike those in categories 4-6, discrepancies in category 3 represent bugs that do not suggest a natural or easy fix.

The fact that only a single inaccuracy bug was discovered in mpmath is unsurprising; mpmath implements arbitrary-precision arithmetic and should therefore be less susceptible to such defects. On the other hand, more than half of the bugs from this discrepancy category (8 out of 13) were attributed to jmat. This might be a consequence of the relative immaturity of the software with respect to the others it was tested against (the first commit to jmat repository was in 2014) and its smaller group of contributors.

**4.3.5 Category 2 Discrepancies.** The following example shows two discrepancies for the same equivalence class containing function synonyms implementing the *exponential integral* or *En-function*:

(GSL)	gsl_sf_expint_En(-2, -2) ⇒ EXCEPTION
(mpmath)	expint(-2, -2) ⇒ -1.84...
(SciPy)	expn(-2, -2) ⇒ inf
(GSL)	gsl_sf_expint_En(1, -1) ⇒ -1.89...
(mpmath)	expint(1, -1) ⇒ EXCEPTION
(SciPy)	expn(1, -1) ⇒ inf

The exception thrown by GSL came with the message `domain error` while the exception thrown by mpmath actually came from the driver around the `expint` function that complained when the return value was a complex number. Coming to conclusions about this particular set of category 2 discrepancies requires some domain-specific knowledge and a little experimentation.

The *En-function* has a branch cut along the negative real axis so for that portion of its domain, these developers made different choices about which value to return. When consulting the documentation, SciPy is the only library that provides an expected domain, stating that both arguments should be greater than or equal to zero. Furthermore, in the examples given in the documentation, they demonstrate that the expected return for inputs outside of the domain is a NaN. As a result, though the differing values are not necessarily buggy, reports were filed for the lack of documentation for GSL with respect to an expected domain, and the inconsistency with which SciPy handles unexpected inputs.

**4.3.6 Category 1 Discrepancies.** Discrepancies in this category are unlikely to represent reportable bugs, though they do illustrate the ways in which intentional choices made by developers can create points of divergence. Consider the following example of a category

1 discrepancy discovered by FPDIFF in the equivalence class of functions implementing the Bessel function of the second kind:

(GSL)	gsl_sf_bessel_Yn(0, -0.5) ⇒ EXCEPTION
(jmat)	bessely(0, -0.5) ⇒ EXCEPTION
(mpmath)	bessely(0, -0.5) ⇒ EXCEPTION
(SciPy)	yn(0, -0.5) ⇒ NaN

The exceptions from jmat and mpmath are from their generated drivers complaining about complex returns and the GSL exception is a `domain error`. From this, we observe that while jmat and mpmath provide support for complex numbers, GSL and SciPy do not. Furthermore, the means by which GSL and SciPy developers choose to handle such computations outside of their chosen scope differs, i.e., throwing exceptions versus generating NaN values.

**RQ3:** FPDIFF discovered 655 unique discrepancies between function synonyms encompassing all six categories of numerical discrepancies. Different sources of adversarial inputs vary in their effectiveness and the type of discrepancies they reveal. Special-value inputs were particularly effective and test-migration inputs proved to be superior to those gathered via a binary guided random search.

**RQ4:** Ignoring category 1 discrepancies, we found that about 46% (150/327) of discrepancies concerned reportedly buggy behavior. These 150 discrepancies represented 125 unique bugs. While identifying such discrepancies can be labor intensive and sometimes requires domain-specific knowledge, those of higher category or larger maximum  $\delta$ -difference values are more easily diagnosed as buggy.

## 4.4 Threats to Validity

In this paper, we implement systematic and automated differential testing to discover discrepancies across numerical libraries. However, our approach has a number of limitations. *First*, our approach only targets a subset of the functions present in numerical libraries. Even so, special functions make up a large class of complex functions that are widely-studied in the literature. The results of our experiments stand on their own. *Second*, our extractor strategy for libraries with incomplete type information is limited by the thoroughness of developer-written tests. We aimed to reduce this threat by including popular numerical libraries that are actively used and developed. *Third*, the choice of elementary inputs can cause the classifier to miss some mappings. We attempted to increase our recall by running the pipeline multiple times and refining the range of inputs heuristically, to positive results. *Fourth*, not all discrepancies represent buggy behavior; results must be manually inspected to determine whether or not a discrepancy is a bug. We take strides to remedy this by proposing a categorization that can be used to effectively triage discrepancies; by automatically removing category 1 discrepancies, we reduce the number to be inspected by half.

## 5 RELATED WORK

*Numerical Software Testing.* A variety of tools have been developed for testing numerical software, from which a large number deal with input generation. Fu and Su [21] and Bagnara et al. [5] generate inputs that maximize code coverage via unconstrained programming and symbolic execution, respectively. Others maximize the floating-point error in the result via genetic algorithms [49], binary search over a specified domain [16], and symbolic execution [25]. In addition to maximizing error, recent work [48] also applies a patch of piecewise quadratic functions to approximate the desired behavior. Barr et al. [7] leverage symbolic execution to discover inputs that trigger exceptions. Chiang et al. [15] discover diverging results in certain types of numerical programs. All the above techniques are complementary to our approach.

To address the missing oracle problem, Chen et al. [12] leverage invariant properties referred to as "metamorphic relations". This technique has seen success in testing certain numerical routines. For instance, LAPACK [2] runs compile-time tests for their Linear Equation and Eigensystem routines using such metamorphic relations. However, such relations require domain-specific knowledge and cannot possibly cover all corner cases. Kanewala and Bieman [28] apply machine learning for mining metamorphic relations.

Other approaches focus on analysis of numerical code. Bao and Zhang [6] use a bit tag to track the propagation of inaccuracies through a program's execution. Benz et al. [9] conduct a parallel "shadow execution" of a program with higher precision "shadow values" to detect inaccuracies. Fu et al. [20] automatically perform backward error analyses on numerical code. Fu and Su [22] propose a floating-point analysis via a reduction theory from any given analysis objective to a problem of minimizing a floating-point function.

When it comes to testing numerical libraries (as opposed to individual functions, either standing alone or within a larger program context), many of the above works and many others like [33, 41, 42] evaluate the effectiveness of their tools on portions of popular numerical libraries such as GSL and LAPACK, but none has applied a differential approach to compare such libraries.

Of the papers cited in this work, only Dutta et al. [18] focus their efforts on numerical libraries written in Python, namely the probabilistic programming libraries Edward and Stan. Though the work of Yi et al. [48] focuses on 20 special functions in GSL, they do have a small aside that applies their tool to 16 analogous functions discovered manually in SciPy. In addition, they also use analogous mpmath functions executed at high-precision as their ground truth.

*Differential Testing.* Differential testing [34] tests multiple implementations that share the same functionality with the same input. The goal is to identify bugs by observing the output differences. Its utility as a tool for finding bugs has been well-documented, being used to test compilers [40, 46], JVM implementations [13], program analyzers [29], probabilistic programming systems [18], interactive debuggers [31], code coverage tools [47], and certificate validation in SSL implementations [14] to name a few.

In its built-in tests, SciPy conducts a manually written differential test of a subset of special functions against the analogous functions in the mpmath library [26]. This manual effort highlights the value of differential testing in the context of numerical libraries.

To the best of our knowledge, we are the first to conduct automated differential testing of numerical libraries for the purpose of discovering discrepancies among libraries.

*Test Migration.* Behrang and Orso [8] introduce a tool for automatically migrating tests across similar mobile applications, essentially translating a sequence of events into a form that can be consumed by the target app. Qin et al. [38] present TestMig, an approach to migrate GUI tests from iOS to Android. To the best of our knowledge, we are the first to use test migration in the context of numerical software.

*API Mapping.* Nguyen et al. [35, 36] use supervised learning to train a neural network on existing code migrations for API mapping. Bui [11] proposes to use Generative Adversarial Networks to "align" two different vector spaces of embeddings generated separately for a pair of APIs. Recent work [17] creates function embeddings to find function "synonyms" across Linux file systems and drivers. However, to the best of our knowledge, such techniques have not been applied to the APIs of numerical libraries. The former approach suffers from a lack of parallel corpora. The latter two utilize embeddings of functions based on their calling context which, for numerical functions, can be widely-varied or even non-existent if being used for one-off calculations.

In the realm of mapping math APIs, Santhiar et al. [39] similarly leverage the idea of mining function specifications from unit tests and using the outputs of functions to produce function mappings. However, their tool MathFinder requires the user to input a description of the desired functionality. This query is then tested against the unit tests of the target library to find matching outputs. MathFinder's motivation is different; mappings are targeted and queries are formed one at a time manually. By contrast, our technique is automatic, untargeted, and it does not require input specifications.

## 6 CONCLUSION

We proposed an approach for finding discrepancies between synonymous functions in different numerical libraries as a means of identifying incorrect behavior. Our approach automatically finds such synonymous functions, synthesizes testing drivers, and executes differential tests to determine meaningful discrepancies across numerical libraries. We implemented our approach in a tool named FPDIFF, which automatically discovered 126 equivalence classes across the libraries GSL, jmat, mpmath, and SciPy with a 95.8% precision and 79.0% recall. The fact that discrepancies are inherently difficult to find enabled us to cluster function synonyms based on their outputs over *elementary inputs*. Using a selection of *adversarial inputs*, we discovered 655 unique discrepancies within equivalence classes of these function synonyms, 150 of which represent 125 unique bugs. We have reported all bugs to library developers; so far, 30 bugs have been fixed, 9 have been found to be previously known, and 25 more have been acknowledged by developers.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1750983.

## REFERENCES

- [1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), 1–84. <https://doi.org/10.1109/IEEEESTD.2019.8766229>
- [2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing '90)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2–11. <http://dl.acm.org/citation.cfm?id=110382.110385>
- [3] Douglas N Arnold. 2000. The Explosion of the Ariane 5. (2000). <http://www-users.math.umn.edu/~arnold/disasters/ariane.html>
- [4] Douglas N Arnold. 2000. The Patriot Missile Failure. (2000). <http://www-users.math.umn.edu/~arnold/disasters/patriot.html>
- [5] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. 2013. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18–22, 2013*. IEEE Computer Society, 1–10. <https://doi.org/10.1109/ICST.2013.17>
- [6] Tao Bao and Xiangyu Zhang. 2013. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 817–832. <https://doi.org/10.1145/2509136.2509526>
- [7] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 549–560. <https://doi.org/10.1145/2429069.2429133>
- [8] Farnaz Behrang and Alessandro Orso. 2018. Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, May 27 – June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 384–385. <https://doi.org/10.1145/3183440.3195019>
- [9] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 453–462. <https://doi.org/10.1145/2254064.2254118>
- [10] Phelim Boyle and Alex Potapchik. 2006. Application of high-precision computing for pricing arithmetic asian options. In *Symbolic and Algebraic Computation, International Symposium, ISSAC 2006, Genoa, Italy, July 9–12, 2006, Proceedings*, Barry M. Trager (Ed.). ACM, 39–46. <https://doi.org/10.1145/1145768.1145782>
- [11] Nghi D. Q. Bui. 2019. Towards zero knowledge learning for cross language API mappings. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*, Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan (Eds.). IEEE / ACM, 123–125. <https://dl.acm.org/citation.cfm?id=3339722>
- [12] Tsong Yueh Chen, Fei-Ching Kuo, T. H. Tse, and Zhiqian Zhou. 2003. Metamorphic Testing and Beyond. In *11th International Workshop on Software Technology and Engineering Practice (STEP 2003), 19–21 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 94–100. <https://doi.org/10.1109/STEP.2003.18>
- [13] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1257–1268. <https://doi.org/10.1109/ICSE.2019.00127>
- [14] Yuting Chen and Zhendong Su. 2015. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 793–804. <https://doi.org/10.1145/2786805.2786835>
- [15] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2015. Practical Floating-Point Divergence Detection. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9–11, 2015, Revised Selected Papers (Lecture Notes in Computer Science)*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.), Vol. 9519. Springer, 271–286. [https://doi.org/10.1007/978-3-319-29778-1\\_17](https://doi.org/10.1007/978-3-319-29778-1_17)
- [16] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15–19, 2014*, José E. Moreira and James R. Larus (Eds.). ACM, 43–52. <https://doi.org/10.1145/2555243.2555265>
- [17] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. See [30], 423–433. <https://doi.org/10.1145/3236024.3236059>
- [18] Saikat Dutta, Owolabi Legunse, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. See [30], 574–586. <https://doi.org/10.1145/3236024.3236057>
- [19] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 509–519. <https://doi.org/10.1109/ASE.2017.8115662>
- [20] Zhoulai Fu, Zhaojun Bai, and Zhendong Su. 2015. Automated backward error analysis for numerical code. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25–30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 639–654. <https://doi.org/10.1145/2814270.2814317>
- [21] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 306–319. <https://doi.org/10.1145/3062341.3062383>
- [22] Zhoulai Fu and Zhendong Su. 2019. Effective floating-point analysis via weak-distance minimization. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 439–452. <https://doi.org/10.1145/3314221.3314632>
- [23] Amparo Gil, Javier Segura, and Nico M. Temme. 2007. *Numerical methods for special functions*. SIAM. <https://doi.org/10.1137/1.9780898717822>
- [24] Brian Gough. 2009. *GNU Scientific Library Reference Manual - Third Edition* (3rd ed.). Network Theory Ltd.
- [25] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution (To Appear). In *International Conference on Software Engineering (ICSE)*.
- [26] Fredrik Johansson et al. 2013. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. <http://mpmath.org/>.
- [27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–). <http://www.scipy.org/>
- [28] Upal Kanewala and James M. Bieman. 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4–7, 2013*. IEEE Computer Society, 1–10. <https://doi.org/10.1109/ISSRE.2013.6698899>
- [29] Christian Klinger, Maria Christakis, and Valentin Wüstholtz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 239–250. <https://doi.org/10.1145/3293882.3330553>
- [30] Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). 2018. *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04–09, 2018*. ACM. <http://dl.acm.org/citation.cfm?id=3236024>
- [31] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers. See [30], 610–620. <https://doi.org/10.1145/3236024.3236037>
- [32] A. D. MacDonald and Sanborn C. Brown. 1949. High Frequency Gas Discharge Breakdown in Helium. *Phys. Rev.* 75 (Feb 1949), 411–418. Issue 3. <https://doi.org/10.1103/PhysRev.75.411>
- [33] Osni Marques, Christof Vömel, James Demmel, and Beresford N. Parlett. 2008. Algorithm 880: A testing infrastructure for symmetric tridiagonal eigensolvers. *ACM Trans. Math. Softw.* 35, 1 (2008), 8:1–8:13. <https://doi.org/10.1145/1377603.1377611>
- [34] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. <http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf>
- [35] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2016. Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 756–758. <https://doi.org/10.1145/2889160.2892661>
- [36] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 438–449. <https://doi.org/10.1109/ICSE.2017.47>
- [37] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. <https://doi.org/10.1145/3276517>
- [38] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International*

- Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019.*, Dongmei Zhang and Anders Møller (Eds.). ACM, 284–295. <https://doi.org/10.1145/3293882.3330575>
- [39] Anirudh Santhiar, Omesh Pandita, and Aditya Kanade. 2014. Mining Unit Tests for Discovery and Migration of Math APIs. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 4:1–4:33. <https://doi.org/10.1145/2629506>
- [40] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 203–213. <https://doi.org/10.1145/2884781.2884879>
- [41] Enyi Tang, Earl T. Barr, Xuandong Li, and Zhendong Su. 2010. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12–16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 131–142. <https://doi.org/10.1145/1831708.1831724>
- [42] Enyi Tang, Xiangyu Zhang, Norbert Th. Müller, Zhenyu Chen, and Xuandong Li. 2017. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-Precision Testing. *IEEE Trans. Software Eng.* 43, 10 (2017), 975–994. <https://doi.org/10.1109/TSE.2016.2642956>
- [43] Lode Vandevenne. 2014. jmat: Complex special functions, numerical linear algebra and statistics in JavaScript. <https://github.com/lvandeve/jmat>. (2014).
- [44] Ran Wang, Daming Zou, Xinrui He, Yingfei Xiong, Lu Zhang, and Gang Huang. 2016. Detecting and fixing precision-specific operations for measuring floating-point errors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 619–630. <https://doi.org/10.1145/2950290.2950355>
- [45] Eric W Weisstein. [n. d.]. Special Function. ([n. d.]). <http://mathworld.wolfram.com/SpecialFunction.html>
- [46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. <https://doi.org/10.1145/1993498.1993532>
- [47] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 488–498. <https://doi.org/10.1109/ICSE.2019.00061>
- [48] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *PACMPL* 3, POPL (2019), 56:1–56:29. <https://doi.org/10.1145/3290369>
- [49] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16–24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 529–539. <https://doi.org/10.1109/ICSE.2015.70>

# Testing High Performance Numerical Simulation Programs: Experience, Lessons Learned, and Open Issues

Xiao He\*

Xingwei Wang

Jia Shi

hexiao@ustb.edu.cn

School of Computer and Communication Engineering,  
University of Science and Technology Beijing  
Beijing, China

Yi Liu\*

liuyi@cert.org.cn

National Computer Network Emergency Response  
Technical Team/Coordination Center of China  
Beijing, China

## ABSTRACT

High performance numerical simulation programs are widely used to simulate actual physical processes on high performance computers for the analysis of various physical and engineering problems. They are usually regarded as non-testable due to their high complexity. This paper reports our real experience and lessons learned from testing five simulation programs that will be used to design and analyze nuclear power plants. We applied five testing approaches and found 33 bugs. We found that property-based testing and metamorphic testing are two effective methods. Nevertheless, we suffered from the lack of domain knowledge, the high test costs, the shortage of test cases, severe oracle issues, and inadequate automation support. Consequently, the five programs are not exhaustively tested from the perspective of software testing, and many existing software testing techniques and tools are not fully applicable due to scalability and portability issues. We need more collaboration and communication with other communities to promote the research and application of software testing techniques.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Software testing, Experience, Numerical simulation, High performance computing

### ACM Reference Format:

Xiao He, Xingwei Wang, Jia Shi, and Yi Liu. 2020. Testing High Performance Numerical Simulation Programs: Experience, Lessons Learned, and Open Issues. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3395363.3397382>

\*Corresponding Authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3397382>

## 1 INTRODUCTION

High performance numerical simulation programs (HPNSPs) are a sort of computer programs that are widely used to simulate complex real-world systems on high performance computers according to mathematical and physical models. They enable us to reproduce, analyze, and predict the behavior of a system in physics, chemistry, biology, economics, psychology, and social science. They play a crucial role in modern science and engineering. Many spectacular findings [4, 26, 48, 56] are made with the help of HPNSPs. For example, Qing Peng et al. [48] revealed the formation mechanism of  $\langle 100 \rangle$  dislocation loops, the defects that greatly decrease the strength of alloyed iron, by adopting large-scale molecular dynamics simulation. HPNSPs also contribute to our daily lives from many aspects. For example, they have been applied to weather forecast [6, 45], earth quake analysis [18, 28], and traffic simulation [17].

The quality of these simulation programs is critical to their successful application. If these programs have defects in mathematical models, numerical algorithms, code implementation, and input data, then any conclusion, decision, or scientific finding made from the simulation results may become unreliable.

Software testing has been demonstrated as an effective way of the verification and validation of software programs. The theories, the techniques, and the tools of software testing have been intensively discussed in all mainstream software engineering forums, and they have been successfully applied to various sorts of systems, such as Web and cloud services [14, 58], mobile applications [41, 43, 61], AI systems [36, 47], and compilers [9, 10, 37].

Surprisingly, there is very limited discussion on how to test high performance numerical simulation programs in the software testing community. Existing research efforts on numerical program analysis [5, 7, 30, 38, 44, 55, 59, 66, 68, 70, 71] mainly focus on simple or preliminary math functions. It is unknown for a software testing practitioner how HPNSPs are tested now, what obstacles there are, and whether the testing techniques that are intensively discussed in the software testing community can improve the testing process.

The major objective of this paper is to present our real-world experience of testing high-performance numerical simulation programs from the perspective of software testing practitioners in the context of a cross-discipline research and development (R&D) team who collaboratively work for a national project. The R&D team aims to establish a multi-physics numerical simulation system for nuclear power plants deployed on high performance computers. The system comprises of several numerical simulation programs.

Experts of nuclear physics, material science, applied mathematics, numerical analysis, high performance computing (HPC), computer graphics and software engineering, coming from 10 universities and institutes all-over the country, participated in this project. We joined this team as software experts two years ago, and are responsible for testing these HPNSPs. After struggling for two years, we realized that the testing of HPNSPs is far more complicated than we expected. There are many challenges and pitfalls in practice.

This paper mainly focuses on the following research questions.

- RQ.1** What testing approaches are applicable and effective?
- RQ.2** Where can we find/reuse existing test inputs? Can we randomly generate new test inputs?
- RQ.3** What kinds of test oracles are in use and how to obtain them?
- RQ.4** What are the most frequently occurring bugs?

In our practice, we adopted five testing approaches and found 33 bugs in total. Based on our experience, our major conclusions are summarized as follows. (1) Property-based testing and metamorphic testing are two effective approaches to testing HPNSPs. (2) Domain knowledge is crucial for testing HPNSPs, but how to express and exchange domain knowledge for testing is an open issue. (3) Due to the lack of test cases/inputs, test oracles, and automation, our programs under test are not exhaustively tested from the perspective of software testing. (4) Hindered by many practical issues, e.g., scalability, many cutting-edge techniques are not fully applicable. We believe that software testing community should have more collaboration and communication with other communities and domain experts to develop new or adapt existing testing techniques for complex computation systems.

The rest of this paper is structured as follows. Section 2 introduces the numerical simulation programs under test and their general structures. Section 3 presents our real experience of testing these programs in detail. Section 4 answers the research questions. Section 5 summarizes the lessons learned, open issues, and threats to validity. Section 6 discusses the related work. The last section presents the conclusion.

## 2 BACKGROUND

### 2.1 Units of Analysis

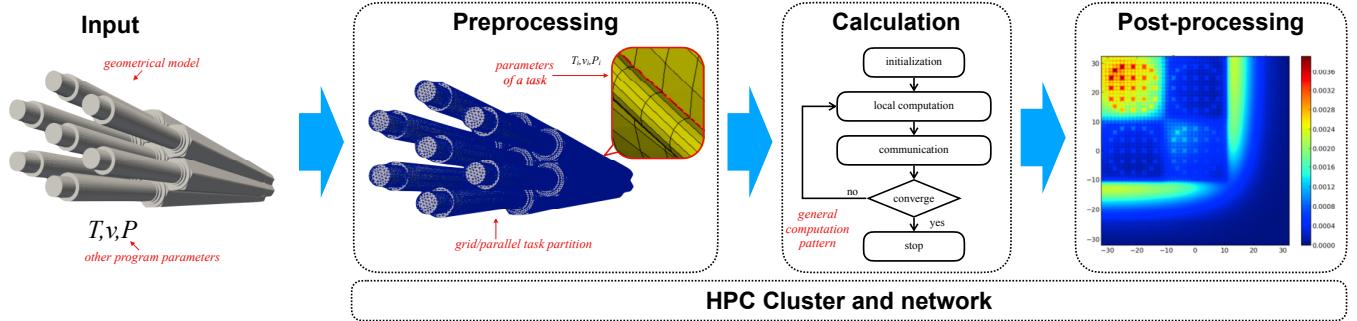
This paper concentrates on the following five high performance numerical simulation programs as the units of analysis in this study because their developers believed they were ready for testing.

- **ANT-MOC** is a simulation program of neutral particle transport based on the method of characteristics. ANT-MOC originates from openMOC [8], but differs from openMOC in the parallelization strategy and the geometrical model supported. ANT-MOC has 31K lines of C++ code and 18 parameters, e.g., a geometrical model that is specified by CAD software, a material data file, and the number of azimuthal and polar angles. Given a geometrical model, ANT-MOC will generate many characteristic lines. Each characteristic line represents the movement trace of a neutral particle. Then, ANT-MOC iteratively updates the movements and states of all particles along these characteristic lines, till result converges. ANT-MOC outputs the spatial distribution of reaction rates that may be visualized as 3D images.

- **MISA-MD** is a molecular dynamics simulation program that is designed and developed by our R&D team. Its functionality is partially comparable with LAMMPS [50]. MISA-MD has 10K lines of C++ code and 16 parameters, e.g., a table file of a potential function, the initial system temperature, and Fe content. At first, MISA-MD will divide the simulation space into many grids and initialize the state of each atom in every grid. Then, for each grid, the program iteratively updates the movement and the state of every atom, and communicates the information of the atoms that move in/out with neighbor grids during each iteration, till the simulation time is reached. The output is the data of atom movement. A 3D animation may be created to show the movement of all atoms.
- **MISA-SCD** is a spatially resolved stochastic cluster dynamics simulation program that computes defect evolution in irradiated materials using a synchronous parallel kinetic Monte Carlo algorithm. MISA-SCD adapts SRSCD [21] for the simulation of I-V-Cu defect species, so SRSCD is not comparable. MISA-SCD has 14K lines of Fortran code and 30 parameters, e.g., a geometrical model file, a material data file, initial temperature, and dislocation density. At first, MISA-SCD will divide the given space into many grids. Then, for each grid, MISA-SCD iteratively updates the defect information and communicates it with neighbor grids, till the simulation time is reached. The outputs are the numbers of defects of every defect species in each grid.
- **MISA-RT** is a deterministic mean-field cluster dynamics simulation program that computes the density of material defects (i.e., clusters). It has 489 lines of C++ code and 4 parameters, i.e., the simulation time, the length of time step, the initial cluster density, and the maximum cluster diameter. At first, MISA-RT generates many ordinary differential equations (ODEs) and sub-divides them into many groups. For each equation group, the program solves the ODEs repeatedly till the simulation time is reached. The output is a density distribution table of material defects.
- **ATHENA** is a simulation program of steady state thermal-mechanical behavior of oxide fuel rods. ATHENA currently has 9K lines of C++ code, and is a rewrite of a Fortran program FRAPCON [1]. It has 132 parameters, including the total number of simulation steps, an array of linear heat generation rates for every step, and the number of rods. First, each rod is split into many grids. For each grid, ATHENA uses a nested loop to realize the calculation, where the inner loop stops when the result converges and the outer loop stops when the simulation reaches a certain number of steps. Finally, the program outputs the temperature, pressure, and deformation of a fuel rod as functions of time-dependent fuel rod power and coolant boundary conditions. The output file may contain more than 20,000 lines of text.

### 2.2 General Structure and Characteristics of High-Performance Simulation Programs

By summarizing the units of analysis, the general structure of high performance numerical simulation programs can be depicted in Fig. 1. Despite of some variations, our units of analysis follow this



**Figure 1: General structure and workflow of high performance numerical simulation**

general structure. As shown in Fig. 1, an HPNSP can be divided into three parts as follows.

- **Preprocessing** module is responsible for dividing the entire computation into a number of parallel tasks (e.g., a characteristics line in ANT-MOC and a grid in MISA-MD and MISA-SCD) and preparing the execution parameters for each parallel task, according to the program input. The program input usually consists of a geometrical model, some physical parameters and constants (e.g., the number of molecules, and the initial pressure and temperature), and some boundary parameters (e.g., the total simulation time). If the geometrical model is available, the commonest way of task partition is to divide the geometrical space into many grids that can be further mapped onto parallel tasks.
- **Computation** module is the implementation of every parallel task that is produced by preprocessing. This module contains several local computation steps and communication steps. At runtime, a process (i.e., an instance of the computation module) is created for each parallel task with its own execution parameters. A process is deployed onto a node of a cluster (or a supercomputer). All processes are run in parallel and communicate mutually via high speed network.
- **Post-processing** module is responsible for gathering the outputs from parallel tasks. Optionally, post-processing visualizes the simulation result using proper graphical representations (e.g., for ANT-MOC and MISA-MD).

According to the general structure above, we identify the following characteristics that are shared by most HPNSPs.

- *The general pattern of the computation module is a loop (or a loop of loops).* The reason is twofold. First, the simulation of a physical phenomenon usually involves a simulation time, and the program updates the system state for each time step till the simulation time is reached (e.g., MISA-MD and MISA-SCD). Second, some numerical algorithms iteratively calculate and refine solutions until convergence errors are lower than a threshold (e.g., ANT-MOC and ATHENA).
- *HPNSP is usually a long-term calculation.* The simulation of complex natural systems requires tremendous computations, even the program is parallelized and deployed on high performance clusters. For instance, ANT-MOC requires at least 2300 core hours to compute a median-sized problem.

- *The computation module may be stochastic.* The randomness mainly comes from the numerical algorithms (e.g., Monte Carlo algorithm in MISA-SCD) that are based on the probability theory. In those programs, computation usually adopts random sampling or decisions. To handle randomness, multiple executions and more computation resources are required.
- *Numerical simulation is by nature inaccurate.* There are many factors besides bugs that contribute to the inaccuracy, including the inaccurate input data, physical and mathematical models, discretization errors and numerical algorithms, floating-point arithmetic, and compiler and library issues. Hence, it is sometimes hard to determine whether or not there is bug in the program when its output is inexact.
- *Program input and output are extremely complex.* The input complexity causes many difficulties in test case development and generation. The following factors contribute to the complexity. First, there are in general a large quantity of input parameters. For instance, MISA-SCD has 30 parameters (see Appendix A) and ATHENA has 132 parameters. Second, some parameters may be very complex in data structure. For instance, ANT-MOC takes a 3D model (just like the model in Fig. 1) as its input and MISA-SCD uses a file to describe composition of a metal material. Third, some parameters are domain specific so that their meanings, ranges, and internal constraints may be unclear to testers. Besides, an HPNSP may output a huge amount of data. For instance, MISA-MD outputs a file for each time step, which contains the information of all atoms at the specific time. In one test, there were 40 time steps and  $2 \times 80^3$  atoms. Consequently, MISA-MD produced over  $4 \times 10^7$  data points and 3GB data as its output. The output complexity causes many difficulties in automated result assertion.

### 3 TESTING PROCESS

When we first joint the project, all the HPNSPs were under development. If domain experts and developers believed that *a program was ready for testing*, then they would share their program with us. Afterwards, we began to design a test plan in collaboration with them. We collected and tried possible testing approaches, test inputs, and test oracles from literature and other members in our R&D team. Then we tested the programs; domain experts and developers also tested the programs by themselves. If a program failed any test,

then its developers were notified. After the developers fixed their program, we continued testing the fixed program.

### 3.1 Testing Approaches

To test the five units of analysis, we first collected the testing approaches that may be used by reviewing the literature on software testing and numerical simulation and by asking the domain experts, the numerical program developers, and the HPC developers in our R&D team *how did you test your program*. Supposing that the program under test (PUT) is denoted as  $p$ , we found the following five approaches that may be applicable in our context.

- **Classical Testing** (CT) refers to the standard testing approach in which  $p$  is fed with a certain input and the actual output is checked by comparing it with an expected result. To apply CT, we must collect adequate test cases, each of which contains a program input and an expected result.
- **Differential Testing** (DT) tests  $p$  by comparing the output of  $p$  with another program  $p'$  that is comparable with  $p$  (or  $p'$  is another implementation of  $p$ ). For a certain test input  $i$ , if  $p(i) \neq p'(i)$ , then a bug is detected. Although the term *differential testing* was proposed by [24, 39], the basic idea had already been widely applied for decades in HPC community who often verify the parallel implementation by comparing the sequential program. DT is also frequently used in the numerical simulation community, in which PUT is often compared with an existing program that is considered well tested. The key to the application of DT is the availability of a well-tested comparable program.
- **Property-based Testing** (PT) refers to the popular testing approach that checks whether the actual output of a single test case  $i$  satisfies a necessary property  $U$  of PUT, i.e., checking  $U(i, p(i)) = \text{true}$ , rather than directly comparing the actual output with the expected. The key to the application of PT is to define a set of high-quality program properties. The properties may be defined according to the theory and the model of the natural system (e.g., Newton's laws), and the expected behavior of the numerical algorithm and parallelization strategy. For instance, *in an isolated physical system, the final kinetic energy is equal to its initial kinetic energy*.
- **Metamorphic Testing** (MT) is a promising testing approach to test case generation and alleviating oracle problem [57, 62]. It has been applied to the testing of numerical programs [11, 19, 20, 64, 65] since it was proposed. The basic idea of MT is to check the satisfaction of a metamorphic relation (MR)  $R$  between a group of source test cases  $i_1, i_2, \dots, i_k$ , a group of follow-up test cases  $i_{k+1}, i_{k+2}, \dots, i_n$  that are generated from the source test cases, and their corresponding output  $p(i_j)$  [13], i.e., checking  $R(i_1, \dots, i_n, p(i_1), p(i_2), \dots, p(i_n)) = \text{true}$ . For instance, for a program that computes the acceleration of an object from a given force (i.e.,  $p(F, m) = F/m$ ), a possible MR is  $F_1 < F_2 \Rightarrow p(F_1, m) < p(F_2, m)$ ; From a source test case  $(F_1, m)$ , according to this MR, we are able to generate many follow-up test cases by increasing  $F_1$ . Similar to PT, MT requires high quality MRs to be used as test oracles.
- **Richardson's Extrapolation** (RE) is used to verify the convergence of numerical programs [51]. Mathematically, if

some desired quantity  $A$  can be computed by an approximation  $A(h)$ , where  $h$  is a step size,  $k$  is a known constant, and  $K$  is an unknown constant, then  $A = A(h) + Kh^k + O(h^{k+1})$  holds. If we ignore the higher order term  $O(h^{k+1})$ , then for  $h$ ,  $h/2$ , and  $h/4$ , it is not difficult to prove that

$$\frac{A(h) - A(h/2)}{A(h/2) - A(h/4)} = \frac{Kh^k - K(h/2)^k}{K(h/2)^k - K(h/4)^k} = 2^k$$

If the program  $p(h)$  correctly implemented the mathematical model  $A(h)$ , then it must satisfy

$$\frac{p(h) - p(h/2)}{p(h/2) - p(h/4)} \approx 2^k \quad (1)$$

RE is a special case of MT and is applicable only if the program involves a step size  $h$  and the constant  $k$  is known.

We believe that a *test case* used in CT is a pair of program input and expected output. For DT, PT, MT, and RE, we also need test cases to execute PUT. In fact, we only used the inputs that are defined in the test cases, because the comparable program (in CT), the properties (in PT), MRs (in MT), and Eq. (1) (in RE) play the role of test oracle. In practice, we both reused the test cases developed for CT, and defined/generated new test cases/inputs if possible. For simplicity, when discussing DT, PT, MT, and RE, we do not differentiate the term *test case* and *test input*.

### 3.2 Overall Results

The general information of the approach application and bugs found is presented in Table 1. We successfully applied four testing approaches to ANT-MOC, MISA-RT, and ATHENA; we applied three approaches to MISA-MD; and we only applied two approaches to MISA-SCD. Hence, each unit was tested using multiple approaches to maximize the possibility of bug detection. Please notice that we worked in a real-world context. We are generally not allowed to freely explore all possible testing approaches and tools due to the limited resource and the time pressure. In practice, we tended to spend more time on the approaches that seemed to be more applicable and effective when testing a program.

CT, DT, and PT were applied to four units, while MT was applied to three and RE was applied to two units. This reflects the usability and applicability of each testing approach. Regarding the possibility of finding bugs, PT and MT revealed bugs in all the units they are applied to; CT seems less likely to detect bugs in our experience.

Table 1 also shows the number of bugs that were found during the testing process. A full list of bugs is available online<sup>1</sup>. A bug in this paper is not a single line of wrong code. Instead, a bug may involve many lines of code that cause the failure of a test. The bugs were counted by the developers who fixed the programs. Presently, we believe that all the bugs were severe because they resulted in incorrect simulation results. It will be the future work to investigate a severity classification for HPNSPs.

In total, we have found 33 bugs, including 7 pre-processing bugs and 24 computation bugs, as shown in Fig. 2. There are also some hidden bugs that fail the test but have not been located. ATHENA has more bugs than others. It is because developers must understand the code of FRAPCON first, and then translate it into the equivalent

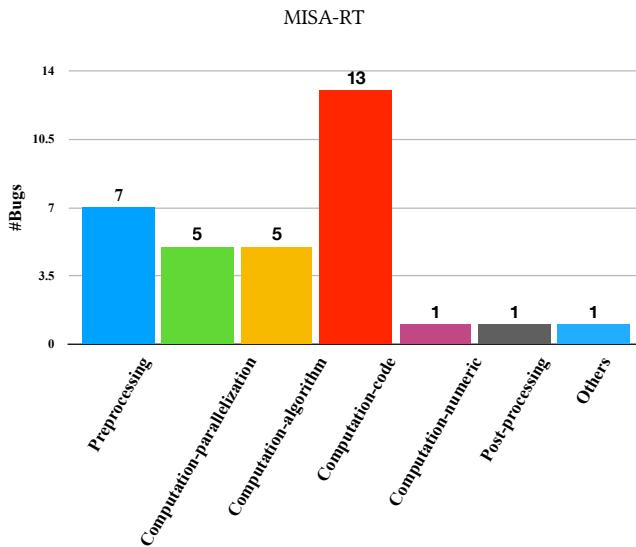
<sup>1</sup>Please visit <https://github.com/ustbsoftlang/issta2020-hpnsp> for the details of all PUTs, test cases/inputs, properties, metamorphic relations, and bugs.

**Table 1: Information of approach application and bugs**

Unit	CT	DT	PT	MT	RE	Total
ANT-MOC	0	1	6	2+	-	9+
MISA-MD	-	0	8	5	-	9
MISA-SCD	?	-	3	-	-	3?
MISA-RT	0	1	-	1	+	1+
ATHENA	3	7+	1+	-	+	11+
Sum	3?	9+	18+	8+	+	33+?

**Notation.** - means that the corresponding testing approach was not applied to the program. Otherwise, the cell tells the bugs found by using the approach: A number indicates how many bugs were already found, and 0 means the program did not fail the tests when an approach was applied; + means that there were some known unrevealed bugs that made the program fail the tests; ? means that we cannot determine whether the failure of the tests was caused by a bug (e.g., it may be attributed to an error in the physical model).

**Note.** The total number of bugs in the last column may not be the sum of the previous columns because a bug may be revealed by multiple approaches.

**Figure 2: Bug distribution****Table 2: Numbers of test cases (#TC), properties (#Pr), metamorphic test groups (#MTG) and metamorphic relations (#MR) used in our practice**

Unit	CT	DT	PT		MT			RE
	#TC	#TC	#TC	#Pr	#MTG	#TC	#MR	#TC
ANT-MOC	6	17	8	5	8	90	3	-
MISA-MD	-	11	4	7	4	17	6	-
MISA-SCD	4	-	4	3	-	-	-	-
MISA-RT	1	3	-	-	1	1000	1	6
ATHENA	1	4	4	2	-	-	-	5

C++ code to develop ATHENA. Many bugs were introduced due to the misunderstanding of the original code.

For all units except ATHENA, PT and MT are more effective in bug detection than CT and DT. DT is very effective for ATHENA

because ATHENA is a rewrite of FRAPCON so that we can apply very fine-grained DT. We will discuss this case in Section 3.3.

Adequate test cases/inputs are essential to systematic testing. To test a program, we developed and applied test cases/inputs as many as possible by reusing existing test cases/inputs and generating new test cases/inputs (we discussed this issue in Section 4.2 in detail), under the restriction of the oracle problem, human efforts, and computation costs. Because the test case/input may contain tens of domain-specific parameters and complex geometry model, as discussed in Section 2.2, it is infeasible to present and explain the definition of any concrete test case in the paper. Please refer to our online data<sup>1</sup> for more information.

Table 2 shows the numbers of test cases/inputs (#TC), properties (#Pr), metamorphic test groups<sup>2</sup> (#MTG), and metamorphic relations (#MR) that were used in the testing process. For all units except MISA-RT, we can only create a small number of test cases. Specifically, in the process of CT, we created a very limited amount of test cases because of the complexity of program inputs and the difficulty in obtaining test oracles; in the process of DT, we used more test cases, but double efforts might be required to provide test cases for both PUT and its comparable program; MT allows us to generate follow-up test cases (inputs only) automatically, and the major restriction is the computation cost—performing more test cases requires more computation time that may not be affordable.

Table 2 also shows the number of properties and metamorphic relations used. We defined 17 properties for 4 units and 10 MRs for 3 programs. Compared with metamorphic relations, it is easier for us to define program properties. The reason is twofold: (1) high-quality metamorphic relations must be derived from physical and mathematical models, but it was beyond our capability; (2) it is more difficult for domain experts and HPC developers to understand the idea of metamorphic testing, so they did not provide much help.

### 3.3 Application of Testing Approaches

This section presents our practice of testing the five units of analysis with the five testing approaches.

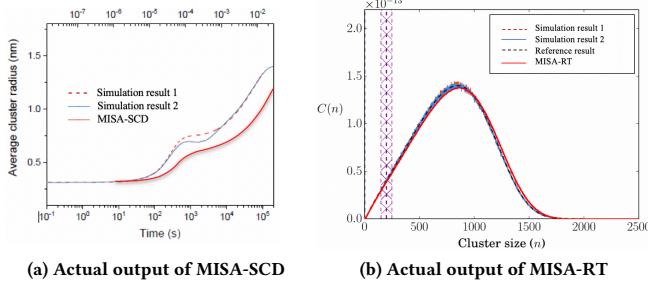
**Application of Classical Testing.** We applied CT to the testing of ANT-MOC, MISA-SCD, MISA-RT, and ATHENA. To perform CT, we need test cases that have both inputs and expected outputs.

Due to the input/output complexity, we cannot construct test cases, especially test oracles, from scratch. Domain experts advised that we could search test cases from domain benchmark problems, literature, and existing programs.

- For ANT-MOC, we found six benchmark problems [22, 32, 63] in the domain of neutral particle transport and nine test cases from openMOC. We developed six test cases from the domain problems, where the expected outputs were formulated by  $k_{eff}$ <sup>3</sup> values. We did not reuse the test cases of openMOC because they are either (the simplified version of) the benchmark problems or lacking oracles.
- For MISA-SCD, we found two test cases from literature [49], where the expected results were presented as diagrams. We

<sup>2</sup>A metamorphic test group is comprised of a set of source test cases and the follow-up test cases that are derived from the source test cases.

<sup>3</sup> $k_{eff}$  is a statistical value calculated from the program output, which is an indicator of the reaction rate. The higher  $k_{eff}$  is, the faster the reaction is.



(a) Actual output of MISA-SCD

(b) Actual output of MISA-RT

**Figure 3: Checking actual outputs using diagrams**

The solid red curves are the outputs of MISA-SCD (Fig. 3a) and MISA-RT (Fig. 3b); other curves are expected results copied from literature.

also derived two new test cases by densifying the mesh and assuming the outputs to be closer to the expected results.

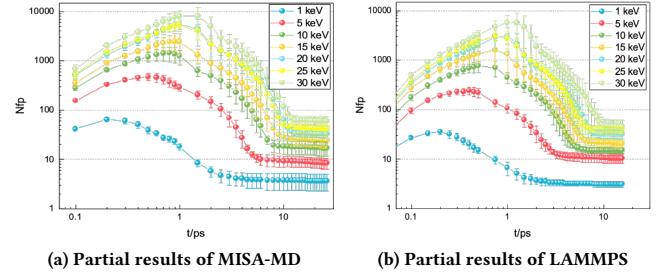
- For MISA-RT, we found one test case from literature [3] whose expected result was also a diagram.
- For ATHENA, we did not find any public test case, so the domain experts created one from their experimental data.

Then, we applied CT and compared the actual outputs with the expected as follows.

- For ANT-MOC, because of the fact that numerical simulation is by nature inaccurate, the actual outputs are not exactly equal to the expected results. For instance, the expected  $k_{eff}$  values of three test cases are 1.128, 1.077, and 1.143, while the actual values are 1.125, 1.074, and 1.141, where the relative errors are less than 0.5%. Both domain experts and developers agreed that the errors are tolerable.
- For MISA-SCD and MISA-RT, because the expected results were defined as diagrams and numeric results are not provided in corresponding literature, we cannot check the program outputs directly. To solve the issue, we drew new curves based on the actual outputs upon the original diagrams, and checked whether the new curves were consistent with the existing ones. Fig. 3 shows two examples of MISA-SCD and MISA-RT, respectively. Fig. 3a shows that the new curve deviates from the expected results of MISA-SCD. However, we cannot judge whether there was a bug. It is because that MISA-SCD adopted a computation method that is different from the method used previously. The deviation might also result from this difference. Fig. 3b shows the curve of the actual output of MISA-RT perfectly fitted the expectation.
- For ATHENA, the major barrier is that the expected output (i.e., the experimental result) is confidential such that it is not shareable. We must send the actual output to domain experts for result checking. Based on domain experts' feedbacks, we found three bugs.

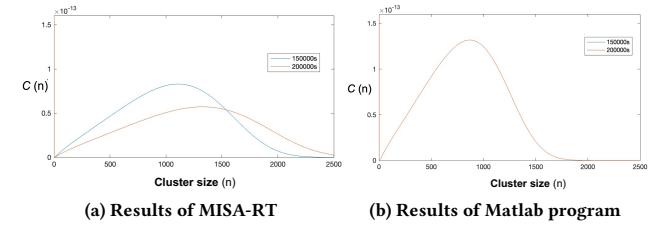
*Application of Differential Testing.* We applied DT to ANT-MOC, MISA-MD, MISA-RT, and ATHENA. To perform DT, we need a program comparable to PUT and some test cases.

- When applying DT to ANT-MOC, we compared the  $k_{eff}$  values of ANT-MOC and openMOC with 17 test cases, including 4 test cases used in CT and 13 derived test cases. We did not



(a) Partial results of MISA-MD

(b) Partial results of LAMMPS

**Figure 4: Differential testing of MISA-MD****Figure 5: Differential testing of MISA-RT**

The two curves in Fig. 5b coincided, while the curves in Fig. 5a did not. This implies that MISA-RT did not reach a fixed pointed as the Matlab program did after the simulation time of 150000 s.

use/create more test cases because for any additional test case, we must prepare two versions for ANT-MOC and openMOC due to their difference of input formats, respectively. For most test cases, the relative error between ANT-MOC and openMOC is very small (< 0.01% on average). There is only one exception, where the difference is 0.21%. From our view, 0.21% is also a small error, but domain experts regarded that the difference is large. Finally, we found a bug of the characteristic line generation in the preprocessing module. After the bug was fixed, the error dropped to 0.042%. As mentioned above, ANT-MOC supports new geometrical models that cannot be handled by openMOC. DT did not cover those problems.

- To apply DT to MISA-MD, we compared MISA-MD and LAMMPS with 11 test cases (containing inputs only) that were developed by ourselves. Because the programs are stochastic, we ran each test multiple times to obtain an average output. Due to the output complexity, as mentioned in Sect. 2.2, it is infeasible to check the outputs of MISA-MD and LAMMPS directly. Hence, we visualized the outputs as figures and animations, and then compared the visual results. For instance, Fig. 4 shows some visual results of MISA-MD and LAMMPS. Obviously, there are significant differences between Fig. 4a and Fig. 4b. However, domain experts regarded that the two figures are similar in their trends and shapes. The divergence was attributed to the different computation methods adopted by MISA-MD and LAMMPS. Finally, MISA-MD passed the tests.

- For MISA-RT, no open-source comparable program existed. To apply DT, we manually developed a sequential Matlab program that realized the same numerical algorithm. We reused the single test case in CT, and the two programs produced the same result. Then, we derived two new test cases by increasing the simulation time. MISA-RT and its Matlab version produced diverse outputs as shown in Fig. 5 which helped us find one floating-point bug about cancellation.
- For ATHENA, because ATHENA is a rewrite of FRAPCON, we conducted a fine-grained differential testing—we compared the output of each function in detail with the single test case used in CT. The domain experts also created several new test cases. They performed DT themselves because the new test cases were not shareable. In this phase, we found seven bugs in total.

*Application of Property-Based Testing.* We applied PT to ANT-MOC, MISA-MD, MISA-SCD, and ATHENA. The key of PT is to identify necessary properties. We defined 17 properties (P1-P17), and a complete list is presented in Appendix B.

Obviously, it is best to define properties according to domain knowledge and physical model. For ANT-MOC, we identified one physical property *P5: the flux distribution must be geometrically symmetric*; For MISA-MD, we identified three physical properties (P9-P11), e.g., *P9: the action and the reaction force between any two molecules must be identical* and *P10: the kinetic energy of the entire system must be conserved*. We also defined a property (P12) for MISA-MD according to domain knowledge, i.e., *P12: within a small time span—two consecutive time steps, the change of the force exerted on a molecule should not be greater than  $\epsilon$ , where  $\epsilon = 20$* . These properties reflect the domain requirements that must be fully satisfied.

However in our practice, it is very hard for us to define the domain and physical properties due to the lack of domain and mathematical knowledge. Hence, we also attempted to find properties from other aspects with which we are more familiar, as follows.

- Four properties (P1-P4) for ANT-MOC and three properties (P6-P8) for MISA-MD were defined based on the implementation, e.g., *P1: the ID of a characteristic line is unique*, and *P7: the positions and the velocities of molecules must not be NAN*.
- Three properties (P13-P15) for MISA-SCD were identified according to the computation method with respect to the output variables Cu, V, SIA\_m, and SIA\_im (i.e., the numbers of Cu atoms, vacancies, mobile SIAs, and immobile SIAs), e.g., *P13:  $\neg(Cu < 0 \wedge V < 0 \wedge SIA\_m < 0 \wedge SIA\_im < 0)$* .
- Two properties (P16-P17) for ATHENA were identified based on the numerical algorithm, i.e., *P16: the pressure of the system should not change within the inner loop*, and *P17: the result of the inner loop must converge*.

We reused the test cases/inputs used in CT/DT to test our programs with the 17 properties. These properties helped us find many bugs, i.e., 6 bugs in ANT-MOC, 8 bugs in MISA-MD, 3 bugs in MISA-SCD, and > 1 bugs in ATHENA. For instance, with the help of P5, we found a bug that the equation  $v' = v/d$  was wrongly realized as  $v' = v/\sqrt{d}$  in MISA-MD. In ATHENA, with the help of P17, we found a bug of the incorrect variability of a variable (i.e., a global variable is declared as a local variable). There were more bugs in ATHENA that broke the properties but have not been located.

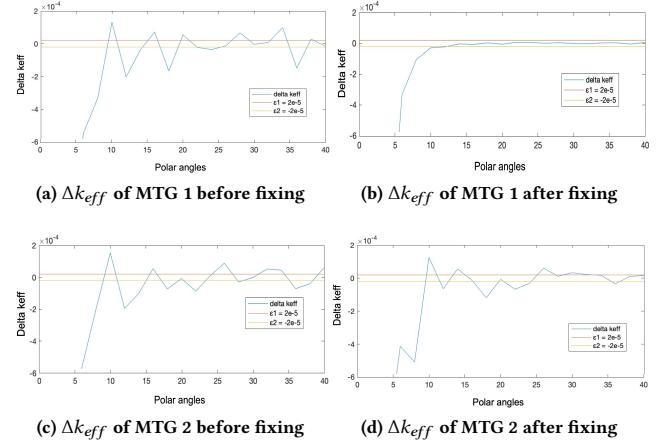


Figure 6: Metamorphic testing of ANT-MOC using MR3

*Application of Metamorphic Testing.* We applied MT to ANT-MOC, MISA-MD, MISA-SCD, and MISA-RT. Assume that  $p$  denotes PUT. Similar to PT, MR identification was challenging. Besides, we found that that *it is more difficult for domain experts to understand the concept of MT*. Hence, they did not help us a lot in this phase. We defined 10 metamorphic relations (MR1-MR10), and a complete list is presented in Appendix C. To perform MT, we reused existing test cases as the source test cases and derived more follow-up test cases according to MRs.

- For ANT-MOC, supposing that the  $k_{eff}$  value of  $p(i)$  is denoted as  $k_{eff}(p(i))$ , we identified three MRs (MR1-MR3), e.g., *MR1: for any two test case  $i$  and  $i'$ , if  $i$  is identical to  $i'$  except that  $i$  is performed on a single physical core and  $i'$  is performed on multiple physical cores, then  $k_{eff}(p(i)) = k_{eff}(p(i'))$* , and *MR3: for a source test case  $i_0$ , if follow-up test cases  $i_1, i_2, \dots$  are derived by gradually increasing the amount of polar angles, then  $|k_{eff}(p(i_k)) - k_{eff}(p(i_{k+1}))| < 2 \times 10^{-5}$  when  $k \rightarrow +\infty$* . MR1 is based on the fact that the number of available CPU cores should not affect the program output; MR3 is based on a property of the computation model. ANT-MOC failed MR1 on Tianhe-2 supercomputer [2] due to the incompatibility of the Intel ICPC compiler. ANT-MOC also failed MR3. Fig. 6a and 6c show the results of two metamorphic test groups. Obviously,  $k_{eff}(p(i_k)) - k_{eff}(p(i_{k+1}))$  did not fall below  $2 \times 10^{-5}$ . We found and fixed one bug in the load balance module. After bug fixing, one metamorphic test group satisfied this MR (Fig. 6b), but the other (Fig. 6d) did not yet. We are still finding the problem now.
- For MISA-MD, we defined six metamorphic relations (MR4-MR9) according to the physical models. For instance, *MR6: if two test cases  $i$  and  $i'$  differ in the amount of Ni atoms only, the final temperatures of  $p(i)$  and  $p(i')$  must be equal*. MISA-MD failed this MR due to a typo in string constants, where “Ni” was written as “Nii”.
- For MISA-RT, we encountered some difficulties during MR identification due to the lack of the domain knowledge, although the program is small. Domain experts only told us

that after a certain time, the physical process will reach a fixed point in reality. We defined an initial MR10 that *if the simulation times of the source and the follow-up test cases  $i$  and  $i'$  are larger than a certain  $T_0$ ,  $p(i) = p(i')$* . However, MR10 is infeasible because neither domain experts nor we know the value of  $T_0$ . To solve this problem, we introduced regression analysis as follows. Given a set of test cases  $i_0, i_1, \dots, i_n$  that only differ in simulation time, find a best fitted function  $g$  from  $p(i_0), p(i_1), \dots, p(i_n)$  and  $g$  must be a function of simulation time. Afterwards, we solve  $dg(t)/dt = 0$ , and the solution is  $T_0$ . MISA-RT did not pass MR10 due to the same bug found by DT.

*Application of Richardson's Extrapolation.* One of the major challenges is the applicability of RE. To avoid misuse, we consulted an expert of mathematics in our university, who identified that RE is applicable to MISA-RT and ATHENA. But for other units, we did not know whether or not RE can be applied due to their complexity.

For MISA-RT, the expert estimated that  $\frac{p(h)-p(h/2)}{p(h/2)-p(h/4)} \approx 16$ , where  $h$  is the length of the time step. However, the actual result was far away from 16—it did not converge. For ATHENA, the variable  $h$  refers to the number of grids, but we did not collect enough results to calculate Eq. (1) before ATHENA crashed. We found that when the number of grids increased over a certain value, some calculations in ATHENA threw expected exceptions. We are still working on bug localization.

RE indicated the presence of bugs in MISA-RT and ATHENA. Nevertheless, it did not provide us with useful information to locate bugs. Eq. (1) is a very strict property, so any tiny error in the computation may cause its violation. We as yet do not know the root causes of the violation of Eq. (1) in MISA-RT and ATHENA.

## 4 ANSWERS TO RESEARCH QUESTIONS

### 4.1 Answer to RQ.1

Based on our experience described in Sect. 3, our answer to **RQ.1** is that CT, DT, PT and MT are relatively easy to be applied, and PT and MT are more effective in bug detection. The virtues and weaknesses of each testing approach are summarized as follows.

CT is the easiest approach that can be learned and applied, especially by the experts outside software engineering. However, its prerequisite that there are sufficient test cases with oracles is not always satisfied. In several cases (e.g., MISA-RT and ATHENA), we could only obtain one test case. The lack of test cases severely affects the usability and effectiveness of CT.

DT is also an easy to understand approach. It can be applied only if a well verified and comparable program is available. However, such a comparable program does not always exist. For instance, MISA-SCD does not have an available comparable program; open-MOC is only partially comparable with ANT-MOC so we cannot use DT to test the new feature of ANT-MOC.

PT and MT alleviate the oracle problems and do not require a comparable program. The major challenge of PT and MT is the identification of program properties and metamorphic relations. This requires a good comprehension of domain theories and knowledge, discretization models, numerical methods, and parallelization strategies. Domain experts, numerical analysis experts, and HPC

developers are highly welcome to join this process. Unfortunately, the concepts of PT and MT (especially for MT) are not very popular outside software engineering community. Extra communication and training costs are required to involve other experts.

RE is very sensitive to bugs. However, RE requires that the mathematical model of the program under test involves a step size  $h$  and that the order  $k$  of the truncate item must be known. Hence, RE is generally not applicable to all HPNSPs. Besides, it does not provide helpful information for bug localization.

Based on our experience and Table 1, we regard that in general PT and MT (and RE is a special case of MT) are more effective than CT and DT, because most bugs were found by PT and MT. The reason why PT and MT are more effective is likely that the properties and the metamorphic relations focus on the expected behaviors of PUT, rather than the program outputs that are by nature inaccurate as discussed in Section 2.2. Properties and relations are more suitable for being test oracles of numerical simulation programs. PT and MT are not comparable because they usually can reveal bugs that were overlooked by each other. To maximize the success rate of bug detection, multiple testing approaches must be applied.

Within the five units, ATHENA is an exceptional case in which DT found most bugs. Because ATHENA is a rewrite of FRAPCON, we are able to compare every function in detail. However, as discussed above, such a comparable program is not always available.

In our practice, we tried several software testing and analysis techniques and tools. Metamorphic testing, as an example of innovative testing techniques, has been successfully applied. However, the application was not plain sailing. As described in Sect. 3.3, we proposed MR10 that is initially infeasible due to an unknown hyper-parameter  $T_0$ . Besides, to test ANT-MOC, we id a MR that *if we input a mirror of the geometric model, then the final result must be symmetric*. The MR was frequently used in literature [19], but was finally proved invalid because ANT-MOC rejects asymmetric geometric model.

We also noticed that many techniques are applicable in theory but infeasible in practice. First, most existing techniques target at C/C++/Java, however Matlab and Fortran are also frequently used in this field. Second, HPNSPs may be run on a supercomputer (e.g., Sunway TaihuLight [29]) that may have a very special hardware/software environment, where most testing tools cannot be used. Third, due to the long execution time, many tools become unscalable. For instance, we tried fpDebug [7], a dynamical floating-point error analysis tool. When fpDebug is turned on, the execution time becomes around 400–500 times slower than that of the original program. Because our programs usually require hours and days to return a result, such dynamic analysis tools cannot be used.

### 4.2 Answer to RQ.2

Based on our experience, our answer to **RQ.2** is as follows: (1) We can find existing test cases/inputs from *domain benchmark problems, existing programs, literature, and experimental data*, though the reusable test cases may be very limited (22 test cases in our practice); (2) Generating new test inputs is possible (but not easy). The sources of test cases and their barriers are summarized as follows.

- *Domain benchmark problems.* Some domains may have defined several benchmark problems (e.g., BEAVRS [32] for

ANT-MOC). However, there are two major barriers to creating test cases from benchmark problems. First, some parameters of the program are described in natural languages, rather than in machine-readable formats. For example, BEAVRS describes that *the reactor core is a pin-by-pin model owning 193 components each of which has  $17 \times 17$  pins*. We exerted CAD software and drew this model manually, which is very time consuming. Second, the problem may not cover all program inputs. Consequently, we must estimate reasonable values for the unspecified parameters.

- *Existing programs*. When there are similar programs, their test cases may also be reusable. Ideally, these test cases can be used to test PUT directly. However in practice, it is very difficult to do so due to the following reasons. (1) The input format of PUT may be very different from that of the comparable program. For instance, the test cases of openMOC were specified using Python scripts, while ANT-MOC adopts a XML-based input format. If we want to reuse test cases of openMOC, then we must manually convert them into XML files. (2) PUT may require some parameters that need to be calculated from the parameters of the comparable program. For instance, MISA-MD requires two parameters  $e_{pka}$  and  $(i, j, k)$  that must be calculated from the parameters  $v_x, v_y, v_z$  of LAMMPS by solving the equation group:

$$\begin{cases} 0.5 \times m_{fe} \times (v_i^2 + v_j^2 + v_k^2) = e2j \times e_{pka} \\ v_i : v_j : v_k = i : k : j \\ v_x = v_i / 100, v_y = v_j / 100, v_z = v_k / 100 \end{cases}$$

where  $e2j = 1.60218 \times 10^{-19}$  and  $m_{fe} = 9.288 \times 10^{-26}$ .

- *Literature*. We used the literature data to test MISA-RT and MISA-SCD. The literature data shares the major barriers of domain benchmark problems and existing programs.
- *Experimental data*. The testing of ATHENA relied on the experimental data. A major barrier in our practice is that the data (and the test cases created) is not fully shareable with us. Consequently, the testing of ATHENA is partially out of our control. Besides, according to the feedback from domain experts, extracting test inputs and expected outputs from the raw data requires tremendous human efforts.

Although we did generate some test inputs in our practice, it is worthwhile emphasizing that test input generation for HPNSPs is not as easy as generating random numbers. First, as discussed in Section 2.2, the input of an HPNSP may be a very complex data structure, e.g., the CAD file of ANT-MOC and the material file of alloy of MISA-SCD. Randomly generating these inputs are challenging. Second, a parameter may have domain specific semantics. In this case, we do not know its range and its impact on computation. Third, some parameters may be correlated under a domain constraint. For instance, MISA-SCD has a parameter that refers to the Cu content in the alloy. If it is increased to a certain threshold value, then the alloy may become another sort and many other parameters must also be altered. Due to these difficulties, we cannot freely generate new test inputs. In practice, we created new test inputs by altering a small subset of the parameter values in the existing ones. Consequently, test adequacy cannot be assured.

### 4.3 Answer to RQ.3

As an answer to RQ.3, we summarized the kinds of test oracles that were used in our practice and their problems as follows.

- *Expected numerical values*. The domain benchmark problems of ANT-MOC include expected numerical values as test oracles. However, these values seldom work because of the inaccuracy and randomness of numerical simulation.
- *Comparable programs*. When DT is applied, the comparable program is used as the test oracle. However, PUT and its comparable program may produce different results if they realized diverse algorithms. On the other hand, the result of the comparable program may also be wrong. These two issues make many difficulties in judging test outputs.
- *Visual results*. The expected outputs may also be depicted as diagrams, images, and even animations. Visual comparison is inevitable in the testing of HPNSPs. First, some literature may only provide the visual results (e.g., MISA-SCD and MISA-RT). Second, due to the output complexity, we may only be able to check the visual results (e.g., MISA-MD). Visual comparison requires extra post-processing efforts. Besides, it also encounters some difficulties in automation, e.g., how to automatically assert that two curves are similar in their trends and shapes could be very challenging.
- *Oracle approximation*. Since the program outputs are by nature inaccurate, an acceptable result range (i.e., oracle approximation [40]) may be used. However, how to determine the bound of the result is another form of oracle problems.
- *Domain experts' opinion*. Domain experts also played a role of test oracles. In the case of ATHENA, a sharp-eyed expert precisely pointed out a tiny error in an 23000-line output file, and helped us find a bug in computation module. However, experts' opinion is not always reliable. When testing MISA-RT, domain experts did not regard the violation of MR10 is due to a bug until we successfully fixed it. Besides, experts may have inconsistent opinions about the same result.
- *Necessary properties*. PT and MT adopt necessary program properties as test oracles. The better we understand the program and the domain knowledge, the better we can define effective properties/MRs. The problem is that learning domain knowledge may be very expensive for testers. For instance, MISA-SCD intended to solve many inter-related equations, which have the form

$$\begin{aligned} \frac{dN_\theta}{dt} = & \tilde{G}_\theta - \sum_\theta \tilde{R}_{\theta\theta'} N_\theta + \sum_{\theta'} \tilde{R}_{\theta'\theta} N_{\theta'} \\ & - \sum_{\theta,\theta'} \tilde{K}_{\theta\theta'} N_\theta N_{\theta'} + \sum_{\theta',\theta''} \tilde{K}_{\theta'\theta''} N_{\theta'} N_{\theta''} \end{aligned}$$

But it is too difficult for a tester to derive any necessary property from this equation.

### 4.4 Answer to RQ.4

Based on our practice, our answer to RQ.4 is that the most frequently occurring bugs are coding errors (13 bugs, such as incorrect variable variability and incorrect index value/constant), and ATHENA contributed most of them (11 bugs). After that, bugs of preprocessing (7 bugs, e.g., the generation of characteristics lines),

bugs of incorrect algorithm implementation (5 bugs, e.g., incorrect implementation of the calculation formula of atom velocities), and bugs of parallel computation and communication (5 bugs) are also very common. Special treatments should be proposed to effectively detect and locate these bugs. Floating-point error (e.g., cancellation) is rare. It does not mean that HPNSPs are free from floating-point errors. As mentioned in our answer to **RQ.1**, the reason may be that we did not find any scalable floating-point analysis tool.

## 5 DISCUSSION

### 5.1 Lessons Learned

We summarized several lessons learned as follows.

- **A proposal for the test strategy for HPNSPs.** Based on our experience, an HPNSP (i.e., PUT) may be tested by the following steps. **Step 1:** apply DT to compare PUT and its sequential version to find parallelization errors. The consistency between the sequential and parallel version is also very helpful for debugging, since parallel programs are much harder to be debugged. If PUT is not developed from a sequential program, then we can treat the execution of PUT that has a single parallel task as the sequential version. **Step 2:** apply PT and MT to verify PUT. Because our practice demonstrated that PT and MT are very applicable and effective, we recommend the two approaches as the primary testing approaches for HPNSPs. **Step 3:** if a comparable program  $p'$  is available, test PUT by comparing with  $p'$ . **Step 4:** validate PUT with domain benchmark problems to address domain experts' concern. The last step is essential because domain experts usually value those domain problems highly (though from the view point of software testing, those problems are generally not sufficient). **In each step**, always test PUT with small-sized test cases that represent simple and primitive physical problems first (we term it *unit data testing* that will be discussed later).
- **Domain experts must be involved in testing.** Frankly, researchers and practitioners of software testing may not be capable to test HPNSPs by themselves. The testing work needs multidisciplinary knowledge, especially the theories and knowledge of application domains. It is impossible for us to quickly manage this knowledge. We must invite domain experts to help us find/develop test inputs and oracles. Nevertheless, domain experts might view this collaboration differently. We regard domain experts as our allies of bug hunting, but they may consider us their employees—we develop and test programs for them. For this reason, domain experts may be unwilling to help us, and this human factor should not be ignored. To improve the present situation, we are trying to establish an industry standard for the testing of nuclear power plant simulation programs, in which the role and obligations of domain experts during the testing process are clearly defined. Regarding technical solutions, we assume that a domain specific language for the specification of domain knowledge should be developed for efficient knowledge transfer across multiple domains.
- **The use of PUT influences the testing.** In our units of analysis, MISA-SCD and MISA-RT are developed based on

new domain theories and models. They will be used to research the theories and the models, rather than making a prediction of the real world. For such programs, it is more difficult to define test oracles because the properties of the theories and models may be unknown yet. Test engineers must be aware of the use of PUT, and discuss the correctness criterion with domain experts before testing.

- **Preserve and manage the actual test outputs seriously.**

Reusing, rather than re-computing, the outputs of an HPNSP will save considerable time and resources. The preserved output data may be used in the following ways. First, when PUT is tested using multiple approaches with the same test case/input, the preserved output can be reused. Second, if the actual outputs of two versions of PUT are preserved, then we can compare them to identify the impacts of the code changes, since a change to HPNSP does not always improve the accuracy. Third, the preserved output may be used for future examination when new result checking methods are available. For instance, ANT-MOC were checked mainly by the metric  $k_{eff}$  for simplicity, and it is our ongoing work to check the outputs by computing *flux distribution*, where the outputs can be reused.

### 5.2 Future Research Suggestions

We must make more research efforts to solve the open issues occurring during the testing of HPNSPs. We identify several potential research directions as follows.

- **Dealing with multi-level errors.** The accuracy of the output of an HPNSP is affected by many factors, e.g., errors in physical/mathematical models, errors in discretization, floating-point errors, parallelization errors, and errors in the input data. When an HPNSP produces an inaccurate output, it is very important to locate the root error to know who is responsible for the incorrect result. Nevertheless, how to analyze multi-level errors is a fresh new topic that requires further investigation.
- **Adapt existing testing techniques and tools for high performance computers and programs.** Many existing software testing techniques and tools are inapplicable to HPNSPs due to scalability issue (i.e., high overhead), the issue of process/thread-unsafe, portability issue (e.g., they cannot be run on a supercomputer), and incompatibility of programming languages (e.g., they are only suitable for C/C++ but not Fortran). There is an urgent need to adapt existing techniques and tools for high performance computing environments and programs. However, doing so is non-trivial. For example, while existing dynamic floating-point error analysis tools [7, 70] usually require >400x longer execution time to conduct the dynamic analysis. Minimizing the overheads of these tools is very challenging.
- **Benchmark development and assessment.** Benchmark problems are intensively used in the community of numerical simulation. However, how to develop and assess benchmark problems is an open issue. Benchmark development requires sufficient domain knowledge, so domain experts are more capable of solving this issue. For benchmark assessment,

- mutation analysis may be helpful. However, it is required to define a set of mutation operators suitable for HNPSPs.
- **Unit data testing.** We observed that a complex test case can be split or simplified into several simpler test cases. For instance, a test case for a full reactor can be simplified into a test case for a single fuel component. Testing with simplest test cases that represent primitive domain problems is termed *unit data testing* in this paper. How to extract/generate simple test cases from a complex one is an interesting topic.
  - **New coverage criterion is required.** Statement coverage is a common indicator of test adequacy. Nevertheless, it may not be useful for testing HPNSPs. For instance, when MISTR was tested, just one test case resulted in 100% statement coverage, but we cannot say the testing with one test case is sufficient. As explained in Section 2.2, the computation module of an HPNSP is usually a loop. Statements in the loop are very likely to be covered during a single execution. New coverage criterion for HPNSPs must be defined to better manage the test progress.
  - **Test case migration.** In the context of differential testing, there is an urgent need for test case migration. Preparing two versions of the same test case is time consuming and error-prone when PUT and its comparable program have distinct input parameters and formats.

### 5.3 Threats to Validity

This paper is based our real experience of testing high performance numerical simulation programs. Due to many real-life restrictions, the paper suffers from the following threats to validity.

- **Internal validity.** A major threat to internal validity is that we may not properly exert the testing approaches due to the limitation of our skill and knowledge. Our conclusion about the approach usability and effectiveness may be affected by this factor. To mitigate this problem, we consulted many famous testing experts of metamorphic testing, mutation testing, and floating-point testing.
- **External Validity.** The major threat to external validity is the limited amount of samples (e.g., programs, test cases, and bugs). It will be our future work to enhance our study when more programs in our team are ready for testing.
- **Construct Validity.** This paper uses the number of bugs to evaluate the effectiveness of a testing approach. There are many other factors that may contribute to the evaluation of the effectiveness. Nevertheless, we believe the number of bugs can be used for qualitative analysis.
- **Reliability.** The presented data might be differently interpreted by others. To mitigate this issue, all the interpretations were thoroughly discussed within the development and testing team.

## 6 RELATED WORK

High performance numerical simulation programs are non-testable. To the best of our knowledge, we are among the first to report real-world experience of testing HPNSPs from the perspective of software testing. Metamorphic testing has been demonstrated to be effective for testing scientific programs, including partial differential

equation solvers, Monte Carlo simulation, bioinformatics programs, and learning and AI systems. Most of these efforts [11, 12, 19, 20, 25, 64, 65] described how to apply MT to and proposed some MRs for the programs under test. There are also some efforts discussing how to identify and infer MRs [13, 35, 69]. However, existing work mainly focused on relatively simple numerical functions, rather than large scale simulation programs.

Floating point analysis attracted much attention [27] during the past decade. Various approaches have been proposed to detect and fix floating point errors [7, 16, 38, 44, 46, 55, 67, 68, 70, 71] and floating-point exceptions [5]. The result of error analysis was also combined with precision turning [15, 33, 42, 53, 54]. The major barrier to their application is the scalability and robustness.

Testing numerical programs is also discussed in the community of numerical simulation. Some efforts focused on reporting the test results of certain programs [23, 31]. Others proposed approaches and tools [51, 52, 60] for verifying numerical programs. Due to the complexity of testing, the numerical simulation community tends to treat errors as an intrinsic characteristic of numerical programs and manage them by uncertainty quantification [34]. These efforts have formed a new research branch of numerical analysis that is independent from software testing. We assume that the two communities should have more collaborations and exchange more experience to improve the testing of numerical programs.

## 7 CONCLUSION

To the best of our knowledge, this paper is among the first to report real-world experience of testing high performance numerical simulation programs from the perspective of software testing. We presented how we applied software testing approaches to five programs in detail. Although we managed to find several bugs for each program under test, we encountered many challenges in test case and oracle development, test automation, and reducing test costs.

We believe that a huge amount of research and engineering efforts are required to make existing testing approaches applicable and more effective to the real-world and complex programs. We also suggest that researchers and practitioners of software testing should have more communication with experts from HPC community and numerical simulation community to share new research achievements with them. We regard this paper as a bridge that brings systematic software testing to numerical simulation.

## ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China (No. 2017YFB0202303) and Beijing Natural Science Foundation (No. 4192036). We thank Prof. TY. Chen from Swinburne University, Prof. Xiaoyuan Xie from Wuhan University, Dr. Zhanwei Hui from PLA Academy of Military Science, Prof. Mingyue Jiang from Zhejiang Sci-Tech University, Prof. Xiaoyin Wang from University of Texas at San Antonio, and Prof. Chang-Jun Hu and Prof. Chang-Ai Sun from University of Science and Technology Beijing for their valuable suggestions on this work during the past two years. We also thank the HPC developers from University of Science and Technology Beijing and the domain experts from China Institute of Atomic Energy for helping us collect and analyze data. Finally, we appreciate the constructive review comments from the anonymous reviewers.

## A PARAMETERS OF MISA-SCD

The following table that is provided by the developers of MISA-SCD lists all the input parameters of MISA-SCD to demonstrate the input complexity of HPNSPs. The table shows that the input of MISA-SCD is complex in terms of the data format, the amount, and the semantics of parameters. Please refer to <https://github.com/ustbsoftlang/issta2020-hpnsp> for more detailed information about the input and output of each HPNPs.

Name	Meaning
defectFile	The defect attributes file.
meshfile	The mesh file.
irradiationType	Type of irradiation ('Cascade' for neutron irradiation, 'FrenkelPair' for electron irradiation, 'None' for no irradiation).
implantScheme	Toggle between Monte Carlo defect implantation and explicit defect implantation.
cascadeFile	The cascade defects file.
implantType	Where uniformly implanting defects.
implantFile	The data file containing non-uniform implantation profile.
grainBoundaries	Toggle whether we are going to include the effect of grain boundaries.
pointDefect	Toggle whether point defects are allowed to move only.
temperature	Temperature (K)
soluteConcentration	Initial content of solute atoms (Cu) in iron.
numVac	Initial number of vacancies in the simulation system.
dpaRate	The rate of DPA.
totalDPA	Total DPA in simulation.
fir	Radiation enhanced factor for Cu.
annealTemperature	Annealing temperature (K).
annealTime	Total anneal time.
lattice	The lattice constant (nm).
burgers	Dislocation loop burgers vector (nm).
reactionRadius	Reaction distances (nm).
grainSize	Grain size (nm).
dislocDensity	Dislocation density ( $nm^{-2}$ ).
cascadeVolume	Volume of cascade ( $nm^3$ ).
max3D	Maximum size for SIA defect to diffuse in 3D.
numSims	Number of times to repeat simulation.
totdatToggle	Toggle whether the total defects file is output (The file contains number of ever defect species, cluster number densities, average cluster size, average radius of clusters and so on).
minSolute	The minimum size of solute clusters included in the statistics.
minVoid	The minimum size of vacancy clusters included in the statistics.
minLoop	The minimum size of SIA clusters included in the statistics.
minVS	The minimum size of S_Vac clusters included in the statistics.

## B ALL PROPERTIES

The following table shows the properties that were used in PT for each program. PID stands for *Property ID*.

Program	PID	Description	Reason
ANT-MOC	P1	The ID of any characteristic line must be unique	Implementation
	P2	The ID of any characteristic line must be a positive integer	Implementation
	P3	The length of any characteristic line must be a positive integer	Implementation
	P4	The ID of FSR must be unique	Implementation
	P5	The flux distribution must be geometrically symmetric	Physics & Computation
MISA-MD	P6	The ID of any molecule must be unique	Implementation
	P7	The position and velocity of a molecule should not be NAN	Implementation
	P8	The position of a molecule must be within the simulation space	Implementation
	P9	The action and the reaction force between any two molecules must be identical	Physics
	P10	The kinetic energy of the entire system must be conserved (equal to the input PKA energy)	Physics
	P11	The number of molecules should not change	Physics
	P12	Within two consecutive time steps, the change of the force exerted on a molecule should not be greater than $\epsilon$ , where $\epsilon = 20$	Domain knowledge
MISA-SCD	P13	$\neg(Cu < 0 \wedge V < 0 \wedge SIA\_m \wedge 0 \wedge SIA\_im < 0)$	Computation
	P14	$\neg(SIA\_m > 0 \wedge SIA\_im > 0)$	Computation
	P15	$\neg((SIA\_m > 0 \vee SIA\_im > 0) \wedge V > 0)$	Computation
ATHENA	P16	The pressure of the system should not change within the inner loop	Numeric algorithm
	P17	The result of the inner loop must converge	Numeric algorithm

Many properties are related to IDs (e.g., P1, P2, and P4), because ID generation in preprocessing module usually may adopt a very complex algorithm, rather than generating a random unique integer.

## C ALL METAMORPHIC RELATIONS

The following table shows the metamorphic relations that were used in MT for each program. MRID stands for *Metamorphic Relation ID*.

Program	MRID	Description	Reason
ANT-MOC	MR1	For any two test case/input $i$ and $i'$ , if $i$ is identical to $i'$ except that $i$ is performed on a single physical core and $i'$ is performed on multiple physical cores, then $k_{eff}(p(i)) = k_{eff}(p(i'))$ .	Execution
	MR2	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing the depth of a control pin, then $k_{eff}(p(i)) > k_{eff}(p(i'))$ .	Physics
	MR3	For a source test case $i_0$ , if follow-up test cases $i_1, i_2, \dots$ are derived by gradually increasing the amount of polar angles, i.e., by densifying the grids, then $k_{eff}(p(i_k)) - k_{eff}(p(i_{k+1})) < 2 \times 10^{-5}$ , when $k \rightarrow +\infty$ .	Computation
MISA-MD	MR4	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing the initial PKA, then $T(p(i)) < T(p(i'))$ , where $T$ extracts the system temperature from the program output.	Physics
	MR5	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing the simulation space, then $T(p(i)) > T(p(i'))$ .	Physics
	MR6	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing Ni content, then $T(p(i)) = T(p(i'))$ .	Physics
	MR7	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing Cu content, then $T(p(i)) = T(p(i'))$ .	Physics
	MR8	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing the initial PKA, then $Fs(p(i)) < Fs(p(i'))$ , where $Fs$ extracts the amount of Frankel defects from the program output.	Physics
	MR9	For a source test case $i$ , if the follow-up test case $i'$ is derived from $i$ by increasing the initial PKA, then $Fp(p(i)) < Fp(p(i'))$ , where $Fp$ extracts the peak value of Frankel defects from the program output.	Physics
	MR10	If the simulation times of the source and the follow-up test cases $i$ and $i'$ are larger than a certain $T_0$ , $p(i) = p(i')$ .	Physics

## REFERENCES

- [1] [n.d.]. <http://www.swmath.org/software/18294>
- [2] 2020. <https://www.top500.org/featured/systems/tianhe-2/>
- [3] Hala Ashi. 2008. *Numerical methods for stiff systems*. Ph.D. Dissertation. University of Nottingham.
- [4] Stefanie L Baker, Aravinda Munasinghe, Bibifatima Kaupbayeva, Nin Rebecca Kang, Marie Certiat, Hironobu Murata, Krzysztof Matyjaszewski, Ping Lin, Cory M Colina, and Alan J Russell. 2019. Puri Fi Cation By Tuning Protein Solubility. *Nature Communications* (2019), 1–12.
- [5] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. *POPL: Principles of Programming Languages* (2013), 549–560.
- [6] Peter Bauer, Alan Thorpe, and Gilbert Brunet. 2015. The quiet revolution of numerical weather prediction. *Nature* 525, 7567 (2015), 47–55.
- [7] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, Vol. 47. ACM Press, New York, New York, USA, 453.
- [8] William Boyd, Samuel Shaner, Lulu Li, Benoit Forget, and Kord Smith. 2014. The OpenMOC Method of Characteristics Neutral Particle Transport Code. *Annals of Nuclear Energy* 68 (2014), 43–52.
- [9] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 223–234.

- [10] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 305–316.
- [11] T.Y. Chen and T.H. Tse. 2002. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings 26th Annual International Computer Software and Applications (COMPSAC'02)*. IEEE Computer Society, Washington, DC, USA, 327–333.
- [12] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. In *BMC Bioinformatics*, Vol. 10, 24.
- [13] Tsong Yueh Chen, Pak Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *Journal of Systems and Software* 116 (2016), 177–190.
- [14] Tsong Yueh Chen, Changai Sun, Guan Wang, Baohong Mu, Huai Liu, and Zhaoshun Wang. 2012. A Metamorphic Relation-Based Approach to Testing Web Services Without Oracles. *International Journal of Web Services Research* 9, 1 (2012), 51–73.
- [15] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *POPL 2017*. ACM Press, New York, New York, USA, 300–315.
- [16] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '14*. ACM Press, New York, New York, USA, 43–52.
- [17] Anthony T Chronopoulos and Gang Wang. 1996. Traffic flow simulation through parallel processing. *Transportation Research Record* 1566, 1566 (1996), 31–38.
- [18] Yifeng Cui, Kim B. Olsen, Thomas H. Jordan, Kwangyoon Lee, Jun Zhou, Patrick Small, Daniel Roten, Geoffrey Ely, Dhabaleswar K. Panda, Amit Chourasia, and et al. 2010. Scalable Earthquake Simulation on Petascale Supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, USA, 1–20.
- [19] Junhua Ding and Xin Hua Hu. 2017. Application of metamorphic testing monitored by test adequacy in a Monte Carlo simulation program. *Software Quality Journal* 25, 3 (2017), 841–869.
- [20] J. Ding, X. Li, and X. Hu. 2019. Testing Scientific Software with Invariant Relations: A Case Study. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 406–417.
- [21] Aaron Y. Dunn, Laurent Capolungo, Enrique Martinez, and Mohammed Cherkaoui. 2013. Spatially resolved stochastic cluster dynamics for radiation damage evolution in nanostructured metals. *Journal of Nuclear Materials* 443, 1 (2013), 128 – 139.
- [22] E E Lewis, M A Smith, N Tsoulfanidis, G Palmiotti, T A Taiwo, R N Blomquist. 2001. Benchmark specification for Deterministic 2-D/3-D MOX fuel assembly transport calculations without spatial homogenisation (C5G7 MOX).
- [23] L Eca and M Hoekstra. 2013. Verification and validation for marine applications of CFD. *International shipbuilding progress* 60 (2013), 107–141.
- [24] Robert B. Evans and Alberto Savoia. 2007. Differential testing: A new approach to change detection, In *ESEC/FSE 2007. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*, 549–552.
- [25] S. M. Yiu F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau. 1998. Application of Metamorphic Testing in Numerical Analysis. In *Proceedings of the IASTED International Conference on Software Engineering*, 191–197.
- [26] Jun-Xuan Fan, Shu-Zhong Shen, Douglas H Erwin, Peter M Sadler, Norman MacLeod, Qiu-Ming Cheng, Xu-Dong Hou, Jiao Yang, Xiang-Dong Wang, Yue Wang, Hua Zhang, Xu Chen, Guo-Xiang Li, Yi-Chun Zhang, Yu-Kun Shi, Dong-Xun Yuan, Qing Chen, Lin-Na Zhang, Chao Li, and Ying-Ying Zhao. 2020. A high-resolution summary of Cambrian to Early Triassic marine invertebrate biodiversity. *Science* 367, 6475 (Jan 2020), 272–277.
- [27] Anthony Di Franco, Hui Guo, and Cindy Rubio-gonzález. 2017. A Comprehensive Study of Real-World Numerical Bug. *ASE'17* (2017), 509–519.
- [28] Haohuan Fu, Conghui He, Bingwei Chen, Zekun Yin, Zhenguo Zhang, Wenqiang Zhang, Tingjian Zhang, Wei Xue, Weiguo Liu, Wanwang Yin, and et al. 2017. 18.9-Pflops Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-Meter Scenarios. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages.
- [29] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (2016), 072001. <https://doi.org/10.1007/s11432-016-5588-7>
- [30] Zhoulai Fu and Zhendong Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*. ACM Press, New York, New York, USA, 306–319. arXiv:[1704.03394](https://arxiv.org/abs/1704.03394)
- [31] S Sara Gilani, H Montazeri, and Bje Bert Blocken. 2016. CFD simulation of stratified indoor environment in displacement ventilation : validation and sensitivity analysis. *Building and Environment* 95 (2016), 299–313.
- [32] Geoffrey Gunow, Benoit Forget, and Kord Smith. 2019. Full core 3D simulation of the BEAVRS benchmark with OpenMOC. *Annals of Nuclear Energy* 134 (2019), 299–304.
- [33] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* 333 (2018), 333–343.
- [34] Jon C Helton, Jay D Johnson, Cedric J Sallaberry, and Curtis B Storlie. 2006. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Engineering & System Safety* 91, 10 (2006), 1175–1209.
- [35] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. 2016. Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach Using Graph Kernels. *Softw. Test. Verif. Reliab.* 26, 3 (May 2016), 245–269.
- [36] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE '19)*. IEEE Press, 1039–1049.
- [37] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (Pittsburgh, PA, USA) (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399.
- [38] Wonyeol Lee, Rahul Sharma, and Alex Aiken. 2016. Verifying bit-manipulations of floating-point. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. ACM Press, New York, New York, USA, 70–84.
- [39] McKeeman W M. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [40] Jinqiu Yang Mahdi Nejadgholi. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *ASE'19*. IEEE/ACM, ACM.
- [41] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *ISSTA'16* (Saarbrücken, Germany) (ISSTA 2016). Association for Computing Machinery, New York, NY, USA, 94–105.
- [42] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2019. ADAPT: Algorithmic differentiation applied to floating-point precision tuning. *SC'19* (2019), 614–626.
- [43] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *Proceedings of the 38th International Conference on Software Engineering (Austin, Texas) (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 559–570.
- [44] David Monniaux. 2008. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems* 30, 3 (may 2008), 1–41. arXiv:[0701192](https://arxiv.org/abs/0701192) [cs]
- [45] George Mozdzynski, Mats Hamrud, and Nils Wedi. 2015. A Partitioned Global Address Space implementation of the European Centre for Medium Range Weather Forecasts Integrated Forecasting System. *The International Journal of High Performance Computing Applications* 29, 3 (2015), 261–273. arXiv:<https://doi.org/10.1177/1094342015576773>
- [46] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*. ACM Press, New York, New York, USA, 1–11. arXiv:[arXiv:1603.09436](https://arxiv.org/abs/1603.09436)
- [47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *GetMobile: Mobile Comp. and Comm.* 22, 3 (Jan. 2019), 36–38.
- [48] Qing Peng, Fanjiang Meng, Yizhong Yang, Chenyang Lu, Huiqiu Deng, Lumin Wang, Suvarnu De, and Fei Gao. 2018. Shockwave generates 100 dislocation loops in bcc iron. *Nature Communications* 9, 1 (2018), 4880.
- [49] Stephanie A. Pitts, Xianming Bai, and Yongfeng Zhang. 2017. Light Water Reactor Sustainability Program Modeling of Cu Precipitate Contributions to Reactor Pressure Vessel Steel Microstructure Evolution and Embrittlement.
- [50] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (March 1995), 1–19.
- [51] Patrick J Roache. 1998. *Verification and Validation in Computational Science and Engineering*.
- [52] Christopher J. Roy and William L. Oberkampf. 2011. A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. *Computer Methods in Applied Mechanics and Engineering* 200, 25–28 (2011), 2131–2144.
- [53] Cindy Rubio-gonz, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious : Tuning Assistant for Floating-Point Precision Categories and

- Subject Descriptors. *SC'13* (2013).
- [54] Cindy Rubio-Gonzalez, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point precision tuning using blame analysis. In *ICSE '16*. 1074–1085.
  - [55] Alex Sanchez-Stern, Pavel Panchevka, Sorin Lerner, and Zachary Tatlock. 2018. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*. ACM Press, New York, New York, USA, 256–269.
  - [56] Felix Schyboll, Uwe Jaekel, Francesco Petruccione, and Heiko Neeb. 2019. Dipolar induced spin-lattice relaxation in the myelin sheath: A molecular dynamics study. *Scientific reports* 9, 1 (2019), 14813.
  - [57] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.
  - [58] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic Testing of RESTful Web APIs. In *Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 882.
  - [59] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić undefined, and Ganesh Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 2 (Dec. 2018), 39 pages. <https://doi.org/10.1145/3230733>
  - [60] A Stamou and Ioannis Katsiris. 2006. Verification of a CFD model for indoor airflow and heat transfer. *Building and Environment* 41, 9 (2006), 1171–1181.
  - [61] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (Paderborn, Germany) (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 245–256.
  - [62] S M Yiu T. Y. Chen S. C. Cheung. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report HKUST-CS98-01. Hong Kong University of Science and Technology.
  - [63] Toshikazu Takeda and Hideaki Ikeda. 1991. 3-D Neutron Transport Benchmarks. *Journal of Nuclear Science and Technology* 28, 7 (1991), 656–669.
  - [64] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. In *ICSE'18*. arXiv:1708.08559
  - [65] Xiaoyuan Xie, Joshua W.K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558.
  - [66] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2017. Automated Repair of High Inaccuracies in Numerical Programs. In *ICSM'17*. IEEE, 514–518.
  - [67] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2018. Efficient Global Search for Inputs Triggering High Floating-Point Inaccuracies. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC 2017-Decem* (2018), 11–20.
  - [68] Xin Yi, Liqian Chen, Xiaoguang Mao, and Tao Ji. 2019. Efficient automated repair of high floating-point errors in numerical libraries. *Proceedings of the ACM on Programming Languages* 3, POPL (jan 2019), 1–29.
  - [69] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-Based Inference of Polynomial Metamorphic Relations. In *ASE'14* (Vasteras, Sweden). Association for Computing Machinery, New York, NY, USA, 701–712.
  - [70] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. 2015. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 529–539.
  - [71] Daming Zou, Muhan Zeng, Yingfei Xiong, Zhoulai Fu, L U Zhang, and Zhendong Su. 2020. Detecting Floating-Point Errors via Atomic Conditions. In *Proceedings of the ACM on Programming Languages*, Vol. 4. 1–27.



# Functional Code Clone Detection with Syntax and Semantics Fusion Learning

Chunrong Fang\*

State Key Laboratory for Novel  
Software Technology  
Nanjing University  
China  
fangchunrong@nju.edu.cn

Zixi Liu

State Key Laboratory for Novel  
Software Technology  
Nanjing University  
China  
lzzcici@qq.com

Yangyang Shi

State Key Laboratory for Novel  
Software Technology  
Nanjing University  
China  
doublesea.shi@qq.com

Jeff Huang

Parasol Laboratory  
Texas A&M University  
USA  
jeffhuang@tamu.edu

Qingkai Shi

The Hong Kong University of Science  
and Technology  
China  
qshiaa@cse.ust.hk

## ABSTRACT

Clone detection of source code is among the most fundamental software engineering techniques. Despite intensive research in the past decade, existing techniques are still unsatisfactory in detecting "functional" code clones. In particular, existing techniques cannot efficiently extract syntax and semantics information from source code. In this paper, we propose a novel joint code representation that applies fusion embedding techniques to learn hidden syntactic and semantic features of source codes. Besides, we introduce a new granularity for functional code clone detection. Our approach regards the connected methods with caller-callee relationships as a functionality and the method without any caller-callee relationship with other methods represents a single functionality. Then we train a supervised deep learning model to detect functional code clones. We conduct evaluations on a large dataset of C++ programs and the experimental results show that fusion learning can significantly outperform the state-of-the-art techniques in detecting functional code clones.

## CCS CONCEPTS

- Software and its engineering → *Functionality; Maintaining software;*

## KEYWORDS

Code clone detection, functional clone detection, code representation, syntax and semantics fusion learning

\*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397362>

## ACM Reference Format:

Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3395363.3397362>

## 1 INTRODUCTION

Code clone detection is fundamental in many software engineering tasks, such as refactoring [2, 4, 20, 33, 34], code searching [1, 24, 26], reusing [6, 12], and bug detection [10, 17]. If a defect is identified in a code snippet, all the cloned code snippets need to be checked for the same defect. As a result, code clones can lead to error propagation, which can severely affect maintenance costs. Thus, code clone detection is substantial in software engineering, and has been widely studied.

Code clone can be divided into four types according to different similarity levels [3]. **Type-1** (Exactly same code snippets): two identical code snippets except for spaces, blanks, and comments. **Type-2** (Rename/parameterize code): same code snippets except for the variable name, type name, literal name, and function name. **Type-3** (Almost identical code snippets): two similar code snippets except for several statements added or deleted. **Type-4** (Functional clones): heterogeneous code snippets that share the same functionality but have different code structures or syntax. Type-1, Type-2, and Type-3 code clones are well detected by many existing approaches [28]. However, there are still some unresolved issues on detecting Type-4 code clones [28]. The Type-4 code clone detection is the most complicated process because the syntax and semantics of source code are flexible. Detecting functional code clones is challenging because the code representations not only need to represent the syntax of source code, but also need to reflect the structure and relationship between code snippets. In order to detect the functional code clones effectively, many approaches try to use syntax-based or semantics-based information to represent source code.

One of the commonly-used syntax-based code representations is AST(Abstract Syntax Tree), which can represent the syntax of each statement. In some cases, code snippets with different syntax

can implement the same functionality, which can be regarded as functional clones. For example, the statement "i = i + 1" and "i += 1" are not identical if we calculate the text-similarity directly. However, these two statements share the same AST. Typical approaches [9, 27] use a syntax parser to parse source code into a syntax tree.

Another commonly-used semantics-based code representation is PDG (Program Dependence Graph). PDG is a graph notation that contains both data dependence and control dependence. PDG can divide code snippets into basic code blocks and characterize their structures. The same functional code snippets with different structures can be detected as clones. For example, the loop structure *for* and *while* have different AST but their PDGs are identical. Typical approaches include Duplix [15] and GPLAG [18] which detect clones by comparing the isomorphic graphs of code pairs.

However, both syntax-based and semantics-based approaches have their limitations. On the one hand, although AST can represent the abstract syntax of source code, it is not able to capture the control flow between statements. On the other hand, each node in PDG is a basic code block (i.e., a statement at least), which is too coarse [36] compared with AST. The improper granularity may lead to a lack of details inside code blocks. For example, for the statement "int a = b + c", the AST nodes are *int*, *=*, *+*, *a*, *b*, *c*. However, this overall statement is stated in a node of PDG. Besides, the PDG for complex codes can be extremely complicated, and computing graph isomorphism is not scalable [18].

Moreover, AST is commonly used to represent a method, while PDG is used to represent an overall file. Thus, most existing approaches compare code similarity at the granularity of a single method or a single file [30, 38]. A functionality may consist of multiple methods. In other words, a single method may not express the complete implication of a functionality. Meanwhile, a file may contain several unrelated functionalities. Consequently, these granularity levels may not be the best choice for functional clone detection.

To overcome the limitations mentioned above, we use the call graph to combine related methods, so the granularity is flexible, and it depends on the granularity of real functionality. Additionally, we use CFG (Control Flow Graph) to represent the code structure instead of the more complicated PDG. CFG can capture the same control dependence as PDG, and it can overcome the time-consuming obstacle. Each method can be generated into an AST and CFG. As the representations of AST and CFG have different structures, they cannot be fused directly. Therefore, we apply embedding learning techniques [7] to generate fixed-length continuous-valued vectors. These vectors are linearly structured, and thus the syntactic and semantic information can be fused effectively. Note that these vectors are particularly suitable for deep learning models.

In this paper, we propose a novel approach to detect functional code clones through syntax and semantics fusion learning. Particularly, we analyze the method call relationship before extracting the syntactic and semantic features. Our basic idea is to identify the similar functionality of different code snippets by analyzing the call graph and combine the syntactic and semantic features based on AST and CFG by embedding techniques [7]. We use AST and CFG to represent the syntactic and semantic features, respectively. With such features, we further train a DNN model to detect functional code clones. We evaluated our approach on a large functional clone

dataset OJClone [22] for C/C++ programs, which contains 104 programming tasks, and each task has 500 source codes submitted by different students. The different source code files for the same task can be regarded as functional clones. The experimental results show that our approach outperforms the state-of-art techniques with F1 value 0.96, including DECKARD [9], SourcererCC [30], CDLH [38], DeepSim [42], and ASTNN [41].

In summary, the main contributions of this paper are as follows.

(1) We propose a fine-grained granularity of source code for functionality identification. We regard the methods with caller-callee relationships as the implementation of a functionality. To the best of our knowledge, our approach first consider call graph in functional code clone detection.

(2) We propose a novel code joint representation with embedding learning of both syntactic and semantic features. With the combination of syntactic and semantic information, functional code clones can be detected precisely.

(3) We present the design and implementation of the proposed approach for C++ programs. Our extensive evaluation on a large real-world dataset shows a promising result on functional code clone detection. We release all the data used in our studies.<sup>1</sup> In particular, we include all the source codes, datasets, code representations, embedding features, and analysis results.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of our approach. Section 3 illustrates our approach in detail. Section 4 shows our experimental evaluation on a large dataset. Section 5 represents the related work of clone detection. Finally, Section 6 concludes this paper.

## 2 BACKGROUND AND MOTIVATION

A typical code clone detection approach usually follows three steps. (1) **Code pre-processing:** remove irrelevant content in source codes, such as comments; (2) **code representation:** extract different kinds of abstractions in source codes, such as AST (Abstract Syntax Tree) [40] and PDG (Program Dependence Graph) [15]; (3) **code similarity comparison:** calculate the distance between two source codes. Code clone is detected when this distance reaches a threshold.

Since the code representation can have a crucial effect on functional code clone detection, we mainly introduce the background of syntactic representation AST (Section 2.1) and semantic representation CFG (Section 2.2). These tree and graph structures cannot be used for networks directly, and we briefly discuss the background of word embedding (Section 2.3) and graph embedding (Section 2.4). The embedding techniques can be used to transform these tree and graph structures into feature vectors.

### 2.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language.<sup>2</sup> This tree defines the structure of source codes. By manipulating this tree, researchers can precisely locate declaration statements, assignment statements, operation statements, etc., and

<sup>1</sup>It is publicly available at: <https://github.com/shiyy123/FCDetector>

<sup>2</sup>[https://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree)

```

1 int A() {
2     int res = 1, i = 0;
3     int len = 10, flag = 100;
4     for(; i < len; i++){
5         res += i;
6     }
7     if (res > flag){
8         res = res - flag;
9     }
10    else{
11        res = flag - res;
12    }
13    return 0;
14 }
15 int D() {
16     //do something
17 }

```

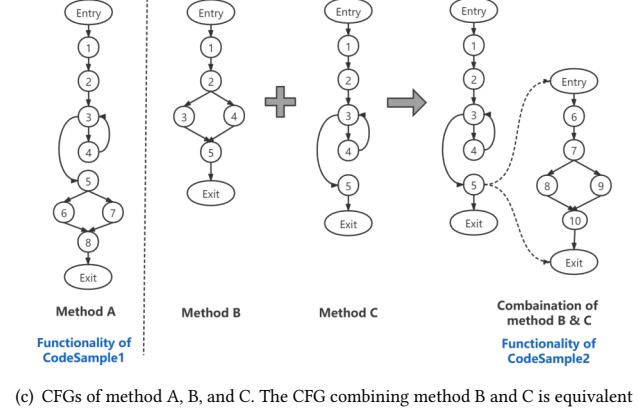
(a) CodeSample1

```

1 int B(int a, int b) {
2     int c;
3     if (a > b){
4         c = a - b;
5     }
6     else{
7         c = b - a;
8     }
9     return c;
10 }
11 int C() {
12     int res = 1, i = 0;
13     int len = 10, flag = 100;
14     while(i < len){
15         res += i;
16         i++;
17     }
18     res = B(res, flag);
19     return 0;
20 }

```

(b) CodeSample2



(c) CFGs of method A, B, and C. The CFG combining method B and C is equivalent to that of A.

Figure 1: Using call graph to identify the functionality

realize operations such as analyzing, optimizing and changing the codes.

Some studies use AST in token-based approaches for source code clone detection [9], program translation [19], and automated program repair [39]. Due to the limitation of token-based approaches [28], these approaches can capture little syntactic information of source code.

## 2.2 Control Flow Graph

Control Flow Graph (CFG) is a representation, using graph notation, of which all paths that might be traversed through a program during its execution.<sup>3</sup> In each CFG, there is a basic block containing several code statements. CFG can easily encapsulate the information per each basic block. It can easily locate inaccessible codes of a program, and syntactic structures such as loops are easy to detect in a control flow graph.

## 2.3 Word Embedding

Word embedding is a collective term for language models and representation learning techniques in natural language processing (NLP). Conceptually, it refers to embedding a high-dimensional space with the number of all words into a continuous vector space with a much lower dimension, and each word or phrase is mapped into a vector on the real number field.

Since the source code and natural language are similar in some ways, many approaches try to use word embedding techniques to process the source code. In our approach, we first traverse each node of AST in preorder and then use word embedding techniques to transform AST into vectors.

## 2.4 Graph Embedding

Graphs, such as social networks, flow graphs, and communication networks, exist widely in various real-world applications. Real graphs are often high-dimensional and difficult to process. Researchers have designed graph embedding algorithms as part of the

dimension reduction technique. Most existing graph embedding techniques aim to transform each node of the graph into a vector. Graph algorithms can be used in different graph tasks, such as graph classification, link prediction, and graph visualization.

Since the CFG is a typical graph, some existing approaches try to use graph embedding techniques for code representation. For example, Tufano et al. [35] uses a kind of graph embedding techniques to represent the semantic features. In our approach, we compare several graph embedding techniques and choose the most effective technique for CFG representation.

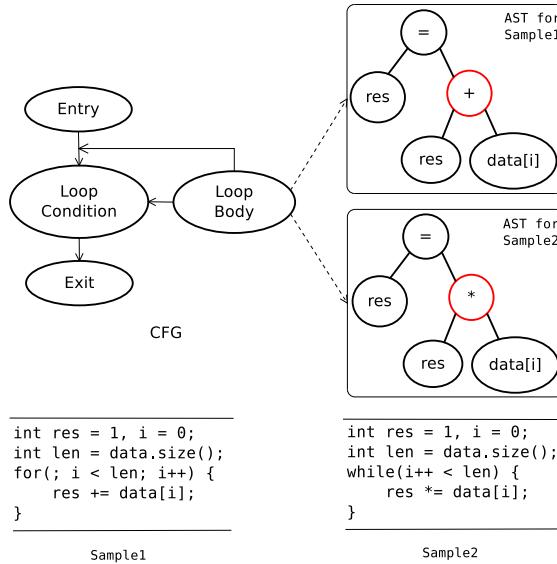
## 2.5 Motivating Example

To detect functional code clones, we first extract the call graph and analyze the functionality of each file. We regard the methods with caller-callee relationships as a functionality. The method without any caller-callee relationship with other methods can be regarded as a self-contained functionality.

For the two source code files shown in Figure 1(a) and 1(b), a method A in *CodeSample1* implements a simple calculation functionality. In *CodeSample2*, a method C calls a method B, and their combination implements the same functionality as the method A. As shown in Figure 1(c), if we detect code clones in method granularity, we can see that the CFGs of method A, B, and C are structurally different. However, after we combine method B and C according to the call graph, we can find the control flow of the combination is equivalent to method A. Therefore, after considering the call graph between methods, we can conclude that method A in *CodeSample1* and the combination of method B and C in *CodeSample2* are functional clones. If we only detect the functionality at method level, the result will be misleading. Besides, it is also inappropriate to detect functional clones in file granularity. Please note that there is a method D in *CodeSample1*, which is irrelevant to the method A, and it is unreasonable to put them together directly. Thus, it is more suitable to extract the call graph and identify the functionality before capturing code features.

After determining the functionalities of each file, we detect functional code clones by analyzing the AST and CFG. CFG can reflect

<sup>3</sup>[https://en.wikipedia.org/wiki/Control-flow\\_graph](https://en.wikipedia.org/wiki/Control-flow_graph)



**Figure 2: AST and CFG for Sample1 and Sample2**

the structure of statements and show the control flow, which is shown in Figure 1(c). The control flow is abstract, and we cannot clearly identify the differences between nodes in CFG. Thus, we detect code clones with syntax and semantics fusion learning. We can capture the syntax by analyzing AST. For instance, if the symbol "+=" in the *for* loop is changed into "\*=" in method A, the functionality is entirely changed. However, since the CFG structure is not changed, it will lead to a false positive result if an approach only takes CFG into consideration. The difference between these two code snippets lies inside the AST of two loop bodies, as shown in Figure 2. Therefore, it is of necessity to consider both syntax and semantics for functional code clone detection.

Moreover, some kinds of code structures can achieve the same functionality, but the AST structures are completely different. For example, both of the *for loop* and *recursive* structure can calculate the factorial of a given input. The ASTs of these two code structures are completely different. However, the inter-procedural control-flow graphs of the two code snippets are very similar, both of which contain a loop. Since our approach not only analyzes the AST, but also utilizes inter-procedural information (such as the call graph), our approach is effective at detecting such inter-procedural code clones.

### 3 APPROACH

As illustrated in Figure 3, our proposed approach consists of the following three components.

**(1) Identify the functionality with call graph.** To identify the functionality in each source code file, we extract the call graph to analyze the caller-callee relationships among methods. The methods with caller-callee relationships are combined together as a functionality. The method without any caller-callee relationship with other methods is regarded as a self-contained functionality.

**(2) Extract syntactic and semantic representations.** In order to capture the code features, we need to extract the AST and

CFG to represent syntax and semantics. First, we extract the AST and CFG of each method. Then, we combine the AST and CFG of related methods according to the call graph. The combined AST/CFG represents a separate functionality. Finally, we use embedding techniques to encode AST and CFG into syntactic and semantic feature vectors.

**(3) Train a DNN model.** To transform the functional clone detection into binary classification, we train a DNN(Deep Neural Network) classifier model with labeled data. When encountered two new functions, we extract and fuse feature vectors and use the trained model to predict whether they are code clones or not.

### 3.1 Identifying the Functionality with Call Graph

We extract the call graph to identify the functionality in each source code file. The call graph represents calling relationships between methods in a program. Each node represents a method, and each edge  $(f, g)$  indicates that the method  $f$  calls method  $g$ . For the input source code files, all statements in them are marked with a globally unique id, which can be regarded as an identifier. The caller-callee relationship is expressed in the form of a triple

$$\langle callerId, statementId, calleeId \rangle.$$

$callerId$  and  $calleeId$  are the statement id of the corresponding method, and  $statementId$  is the id of call statement. For example, in Figure 1(b), the method C calls the method B. Therefore, the caller statement is  $\text{int } C()$  in line 10, the call statement is  $c = B(a, b, flag)$ ; in line 14, and the callee statement is  $\text{int } B(\text{int } a, \text{int } b, \text{bool } flag)$  in line 1. According to the call graph expression, we can obtain the connected methods as a functionality.

To illustrate the caller-callee relationships, we list six call graph cases, as shown in Figure 4. These six cases of static call graph are the most commonly used call graph. Our approach includes but is not limited to these six caller-callee relationships. The caller-callee relationships of methods are explained below. In case (1), there is a functionality that includes method A; in case (2), method A has a recursive call for itself, and there is also a functionality that includes a single method A; in case (3), there exists method A calls method B, and this case has a functionality that includes method A and B; in case (4), method A and method B call each other, and there is a functionality that includes method A and B; in case (5), method A calls method B and method A calls method C, but there is also only one functionality, and the functionality includes method A, B, and C; in case (6), method B calls method A and method C calls method A. There are two functionalities, one including methods  $\{A, B\}$  and another including methods  $\{A, C\}$ .

### 3.2 Extracting Syntactic and Semantic Representations

**3.2.1 AST Representation.** Suppose that  $C$  is a code snippet  $C$  and  $N_{root}$  is its corresponding AST entry node. To extract the syntactic representation of  $C$ , we start from  $N_{root}$  and iterate through all the nodes of AST in preorder. In each AST node, there is an identifier, such as the symbols and variable names. The identifier sequence

$$Seq = \{ident_1, ident_2, \dots, ident_n\}$$

generated in this process can be used to represent the syntactic information of  $C$ .

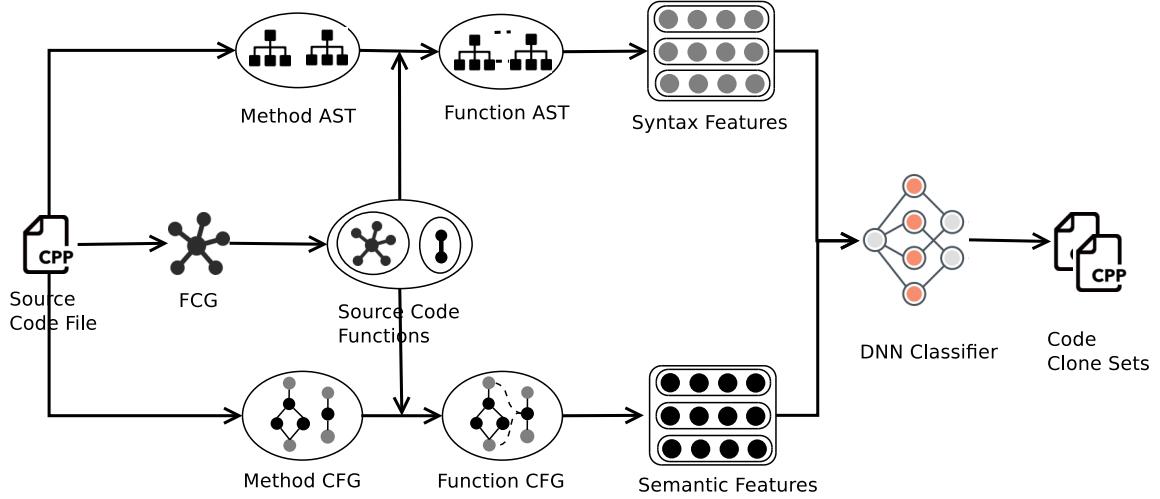


Figure 3: The architecture of our approach

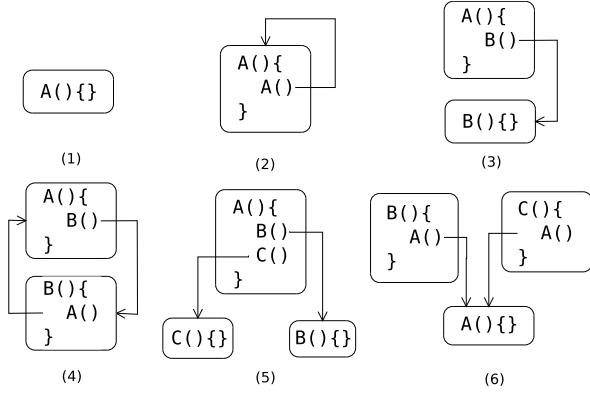


Figure 4: Six kinds of caller-callee relationships among methods

The naming of variables may vary greatly in different programs. Thus variable names can have an effect on functional clone detection. Given an identifier sequence  $Seq$ , we first normalize it by replacing constant values and variables by its type:  $\langle int \rangle$ ,  $\langle double \rangle$ ,  $\langle char \rangle$  and  $\langle string \rangle$ , and add a global self-increasing number, like  $\langle int_1 \rangle$ . When encountered a decimal, no matter it is a  $\langle float \rangle$  or  $\langle double \rangle$ , we unify them as  $\langle double \rangle$ .

We first extract the AST of each method and obtain the corresponding identifier sequence  $Seq$ . Then according to the caller-callee relationships extracted by call graph stated in Section 3.1, we put the AST of each connected method together as a set of AST. Finally, for the methods with connections, we use the set of their AST to represent the syntax of the functionality. Similarly, for the method without caller-callee relationship, we use the corresponding AST to represent the syntax of the functionality.

**3.2.2 CFG Representation.** Compared to AST, CFG captures more comprehensive semantic information such as branch and loop. Although CFG also contains syntactic information at the level of basic

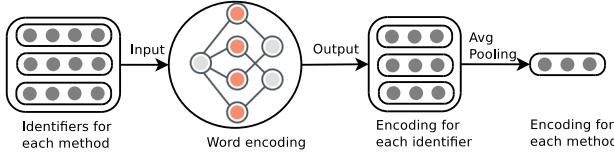
blocks, its representation is too coarse for detecting functional clones compared to the syntactic information extracted from AST at identifier-level granularity. Thus we focus on extracting semantic information from CFG.

Given a code snippet  $C$ , we extract its CFG  $G = (V, E)$  for each method, where  $V$  is a set of vertexes and  $E$  is a set of edges. Each vertex contains a statement of source codes, and the edges indicate the control flow of statements.

We first extract the CFG of each method and get the corresponding graph  $G$ . Then according to the call graph stated in Section 3.1, we connect the CFGs and generate a larger CFG to represent a functionality. The specific connection rules are presented as follows. In case (1), the CFG of a functionality is identical to the CFG of the method; in case (2), the CFG of a functionality is also identical to the CFG of the method, and an edge is added from caller statement to the entry node of the method; in case (3), first we need to add the CFG of method A and B, then we obtain the parent nodes list and the child nodes list of the functionality call statement in method A, and then all the nodes are pointed to the entry node in method B, and all the child nodes point to the exit node in method B; case (4), (5) are similar to case (3), and the edges related to another function call statement are modified, additionally; in case (6), there are two functionalities, so we need to process these two functionalities respectively.

**3.2.3 Embedding Syntactic Feature of Functionality.** Each method in source code is now represented as a sequence of identifiers. We then use Word2vec [21], a word embedding technique, to encode syntactic features. Word embedding is a general term of language model and embeds high-dimensional space words into a much lower dimensional continuous vector space, where each word is mapped to a vector in the real number field.

As shown in Figure 5, in order to encode the syntactic features of each method, the normalized identifier sequence  $Seq$  for each method are put together as the corpus. Word2vec uses the corpus as the input of training model, and output a set of fixed-length vectors to represent each AST node. To get a syntactic feature vector of each



**Figure 5: Syntactic feature embedding**

method, we apply average pooling to all the vectors of identifiers in each method. After this step, the syntactic information of each method is represented as a fixed-length feature vector

$$mv = \text{average}(h_i), i = 1, \dots, N$$

Where  $h_i$  is the feature vector of each identifier.

For each functionality (i.e., the connected methods), we then apply average pooling on all the methods to produce the corresponding syntactic feature. The syntactic information of each functionality is represented as a fixed-length feature vector

$$fv = \text{average}(mv_i), i = 1, \dots, N$$

Where  $mv_i$  is the syntactic feature vector of each method.

**3.2.4 Embedding Semantic Feature of Functionality.** We use graph embedding techniques to encode the CFG. Graph embedding is to use a low dimensional and dense vector to represent every node in the graph. The vector representation can reflect the structure in the graph. In essence, the more adjacent nodes the two nodes share (i.e., the more similar the contexts of the two nodes are), the closer the corresponding vectors of two nodes are. After graph embedding, each method is represented as a fixed-length feature vector. There are many graph embedding techniques, e.g. Graph2vec [23], HOPE [25], SDNE [37], and Node2vec [5].

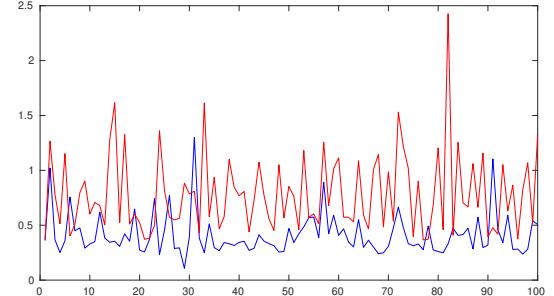
Different from other graph embedding techniques, Graph2vec can generate a vector to reflect the feature of an overall graph. However, other techniques mentioned above can only generate a vector for each node in the graph, and the feature vector of the overall graph is the average value of all nodes. Thus, we use Graph2vec to generate feature vectors instead of the other techniques. We compare different embedding techniques and the results in Table 4 show that Graph2vec performs the best F1 value.

For each method, we use the feature vector generated by Graph2vec to represent the semantic feature vector. For each functionality, we connect all the CFGs of connected methods according to the call graph, which is illustrated in Section 3.1. According to the connection rules stated in Section 3.2.2, we can obtain the connected CFG of connected methods. Then we apply graph embedding on the connected CFG of this functionality. Finally, we transform the connected CFG into feature vectors using the technique of graph embedding.

### 3.3 Training a DNN Model

To detect functional code clones, we may directly compare the Euclidean distance between the joint feature vectors of syntactic and semantic encodings at functionality granularity, as described in the previous section. This distance-based approach has been used in prior work to detect syntactic clones, such as Deckard [9]. However, calculating the distance directly does not work in our case because the distance between the extracted joint feature vectors

is irregular, as illustrated in Figure 6. This is because the weight of each dimension is different, and it is difficult to manually set an appropriate threshold to distinguish between functional code clones.



**Figure 6: Euclidean distance between feature vectors.** The blue line represents the distance between similar code snippets, and the red line represents the distance between dissimilar code snippets. The horizontal axis is the id of code snippets

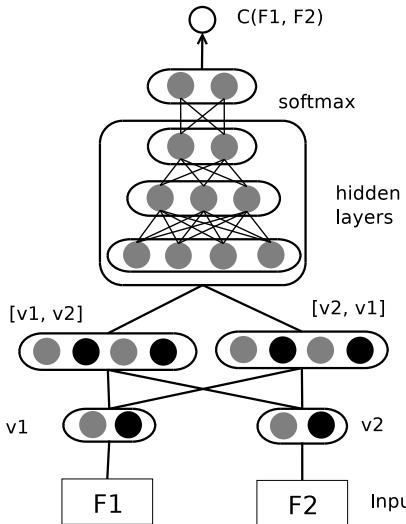
To address this problem, we propose to use a deep learning model that can effectively detect functional clones between code snippets. In particular, we fuse syntactic and semantic feature vectors as input to train a feed-forward neural network [32], and convert code clone detection to binary classification with the softmax layer at the end of the model. Instead of choosing a specific distance metric or set the threshold for clone detection, we adopt deep learning techniques for functional code clone detection that can automatically learn latent code features from the fused feature vectors.

The architecture of the DNN model is shown in Figure 7, which consists of four components. Firstly, we obtain the syntactic and semantic features following Section 3.2.4 and concatenate the two features together as the input of the model, where syntactic and semantic information are both represented with a 16-dimensional vector.

To obtain word embeddings and graph embeddings for AST and CFG, we use Word2vec and Graph2vec to generate word embeddings of length {4, 8, 16, 32}, and graph embeddings of length {4, 8, 16, 32}. We choose the vector length as 16, according to the experiment. In short, the longer vector will increase the training time, and the shorter vector cannot capture sufficient code features. The detailed experiment is illustrated in Section 4.4.

As for the input of the DNN model, we use a pair of code features to represent whether the two functionalities are clones. The input vector contains three portions: the fusion feature vector  $V_1$  for the first functionality, the fusion feature vector  $V_2$  for the second functionality, and a label. The fusion feature vector contains a 16-dimensional syntactic vector and a 16-dimensional semantic vector. The label is a boolean value, where 0 represents non-clone pair, and 1 represents clone pair. Thus, the total dimensions of the input are  $65(32+32+1)$ .

The order of the fusion features of two functionalities,  $[V_1, V_2]$  and  $[V_2, V_1]$ , may affect the classification results. Therefore, we



**Figure 7: The architecture of the deep fusion learning model**

place a fully connected layer before the input of the two features. Then, the hidden layer uses linear transformation followed by a squashing non-linearity to transform the inputs into the neural nodes in the binary classification layer. This step can provide a complex and non-linear hypothesis model, which has a weight matrix  $W$  to fit the training set. The feed-forward step has been completed so far. In the next component, this model then uses back-propagation to adjust the weight matrix  $W$  according to the training set. Finally, there is a softmax layer, which converts the functional clone detection into a classification task.

Suppose there are two code fusion vectors  $V_1$  and  $V_2$ , and their distance is measured by  $d = |v_1 - v_2|$  for syntactic and semantic relatedness. Then we treat the output as their similarity:

$$\begin{aligned}\hat{y} &= \text{softmax}(\hat{x}) \in [0, 1] \\ \hat{x} &= W \mathbf{v} + b\end{aligned}$$

Where  $W$  is the weight matrix of hidden layers. This model uses a cross-entropy loss function, which performs better than mean-squared loss function in updating weight. The goal of training the model is to minimize the loss. We choose AdamOptimizer [13] because it can control the learning rate in a certain range, and the parameters are relatively stable. The model is a DNN classifier and has two classes.

After all the parameters are optimized, the trained models are stored. For new code fragments, they should be preprocessed into syntactic and semantic vectors and then fed into the model for prediction. The output is 1 or 0, which represents clone or non-clone code pairs.

## 4 EXPERIMENTAL EVALUATION

With the implementation of our approach, we conducted evaluations to explore the following research questions.

- RQ1: How does our approach perform in functional code clone detection comparing to state-of-the-art approaches?
- RQ2: How does fused code representations perform in functional code clone detection?

- RQ3: How does our chosen embedding techniques perform in functional clone detection?
- RQ4: How does DNN perform in functional code clone detection comparing to other traditional machine techniques?

### 4.1 Experimental Settings

We used joern to obtain ASTs and CFGs of C++ source code. To extract syntactic features, we trained embeddings of tokens using word2vec with Skip-gram algorithm and set the embedding size to be 16. To extract semantic features, we trained embeddings of graph using Graph2vec and set the embedding size to be 16. There are 5 hidden layers of the DNN classifier, and the dimension of each hidden layers are 128, 256, 512, 256, 128, respectively.

We evaluated our approach on a real-world dataset: OJClone [22] from a pedagogical program online judge system, which mainly includes C/C++ source code. OJClone contains 104 programming tasks, and each task contains 500 source code files submitted by different students. For the same task, the code files submitted by different students are considered as functional code clones. And each code snippet pair can be expressed with  $(s_1, s_2, y)$ , where  $s_1$  and  $s_2$  are two code snippets and  $y$  is the code clone label of them. The value of  $y$  is  $\{0, 1\}$ . If  $s_1$  and  $s_2$  are solving the same task, the value of  $y$  is 0. If  $s_1$  and  $s_2$  are solving different tasks, the value of  $y$  is 1. We choose the first 35 problems from the 104 problems with 100 source files from each problem.

Then, we generated  $100^*100$  clone pairs for each task and  $100^*100$  non-clone pairs for two different tasks. When the problem set is more than two, the non-clone pairs will be far more than clone pairs. The imbalance of data will affect the experimental results, so we adopt under-sample to balance the clone pairs and non-clone pairs. We randomly divide the dataset into two parts, of which the proportions are 80% and 20% for training and testing. We use AdamOptimizer [13] with learning rate 0.001 for training and the number of training epochs is 10000. All the experiments are conducted on a server with 8 cores of 2.4GHz CPU and a NVIDIA GeForce GTX 1080 GPU.

### 4.2 The Effectiveness of Our Approach

*RQ1: How does our approach perform in functional code clone detection comparing to state-of-the-art approaches?* To evaluate the performance of our approach, we compare our approach with several state-of-the-art code clone approaches as follows:

- Deckard, a syntax-tree-based approach, which uses Euclidean distance to compare the similarity between code snippets to detect code clone.
- SourcererCC, a token-based code clone detector for very large codebases and Internet-scale project repositories.
- The approach proposed by White et al. converts source code into the defined tree structure and uses a convolution neural network to learn unsupervised deep features. Later we will name it DLC for short.
- CDLH, a deep learning approach to learn syntactic features of code clone, which converts code clone detection into supervised learning of hash characteristics of source code.
- DeepSim, a semantic-based approach that applies supervised deep learning to measure functional code similarity.

- ASTNN, a state-of-the-art neural network based source code representation for source code classification and code clone detection.

We compare our approach with these code clone detection approaches in terms of Precision (P), Recall (R), and F1-measure (F1). Because our dataset mainly belongs to functional code clones, the detection performance can indicate the ability of these approaches to detect functional clones.

**Table 1: Results on OJClone**

Tools	P	R	F1
Deckard	0.99	0.05	0.10
SourcererCC	0.07	0.74	0.14
DLC	0.71	0.00	0.00
CDLH	0.47	0.73	0.57
DeepSim	0.70	0.83	0.76
ASTNN	0.98	0.92	0.95
<b>Our approach</b>	<b>0.97</b>	<b>0.95</b>	<b>0.96</b>

Table 1 shows the precision, recall, and F1 values of the six approaches. It can be observed that CDLH, DeepSim, ASTNN and our approach can outperform the other three approaches: Deckard, SourcererCC, and DLC in terms of F1 value. As most of the code clones in OJClone belong to functional clones, the results means that CDLH, DeepSim, ASTNN and our approach can well deal with functional clones, while the other three approaches cannot. CDLH, DeepSim and our approach leverage the supervised information to adjust the training process and improve the clone detection model, while the other three are unsupervised approaches. This result means unsupervised approaches can hardly learn the similarity between functional clones. For Deckard, the syntactic information of the source code is embedded into a feature vector, and we calculate the Euclidean distance between feature vectors to detect code clones. In the process of calculating Euclidean distance, the weight of each dimension is considered the same. However, for syntactic feature vectors in functional code clone detection, the importance of each dimension in the feature vector is tend to be different. So the recall of Deckard is pretty low, but the precision becomes very high when the syntactic information of the two code snippets happen to be identical. SourcererCC obtains high recall while getting pretty low precision since it mistakes too many code snippet pairs as code clones. This is because the variable name of these code snippets submitted by students are not standard, and students tend to use simple names like *a*, *b*, and *c*. SourcererCC mistakes these code snippets with the same tokens as code clone. DLC is similar to Deckard and is an approach using the syntactic features to train the convolutional neural network [11] also failed to detect functional code clones without the guidance of supervised information.

Compared with CDLH, a syntax-based deep learning clone detection approach, our approach achieves much higher precision and a little higher recall. Compared with DeepSim, a semantics-based deep learning clone detection approach, our approach achieves higher precision at the sacrifice of a little recall. Compared with ASTNN, an AST-based deep learning clone detection approach, our

approach achieves higher recall at the sacrifice of a little precision. This is mainly because our approach leverages both syntactic and semantic information in functionality granularity. In addition, since our approach utilizes inter-procedural information (such as the call graph), we are good at detecting inter-procedural code clones, which is demonstrated in Section 2.5.

We evaluated the time performance of approaches mentioned above. We ran each tool to detect code clones in OJClone with the optimal parameters. And we run each tool three times and report the average.

**Table 2: Time performance on OJClone**

Tools	Training Time	Prediction Time
Deckard	-	32s
SourcererCC	-	30s
DLC	120s	67s
CDLH	8307s	82s
DeepSim	14545s	34s
ASTNN	9902s	72s
<b>Our approach</b>	<b>8043s</b>	<b>63s</b>

Table 2 reports the time performance of these tools. Deckard and SourcererCC do not need to train the model and conduct clone detection directly. Thus, we use “-” in training time for Deckard and SourcererCC. DLC, CDLH, DeepSim, ASTNN, and our approach need to train the clone detection model. Our approach also needs to conduct embedding learning for syntactic and semantic information. Thus it takes the long time in the training step. Although the training time of DeepSim, CDLH, ASTNN, and our approach are too much compared to the other three approaches, it is a one-time offline process. Once the model is trained, it can be reused to detect code clones. DTC incorporates a simple deep learning model, which cost much less time. Compared with other approach, especially ASTNN, which is the state-of-the-art approach, the training time of our approach is much shorter. Thus, our approach is more efficient. In the long run, the use of deep learning is more convenient, and the trained model can be reused to detect code clones.

### 4.3 The Effectiveness of Fused Code Representations

*RQ2: How does fused code representations perform in functional code clone detection?* In the previous section, we have validated the superiority of our approach on functional clone detection. In this section, we further validate the effectiveness of the syntax and semantics fusion learning.

For this purpose, three different code representations, including text, AST, and CFG, are used to compare the detection performance. Code tokens is also a common representation of source codes. However, AST is a higher abstraction of code tokens and contains more syntactic information than code tokens<sup>4</sup>. And we do not choose code tokens in our experiment. To extract the text sequence of

<sup>4</sup><https://stackoverflow.com/questions/25049751/constructing-an-abstract-syntax-tree-with-a-list-of-tokens/>

source codes, we use the Turing eXtender Language proposed in NICAD [29], a famous text-based clone detection approach. The extraction of AST and CFG has been detailed in the previous section.

Given a source code snippet and three code representations  $R = \{R_1, R_2, R_3\}$ , where the three representations are text, AST and CFG in turn. Word2vec is used to conduct embedding learning for the first two representations and Graph2vec for CFG. Then the embeddings  $E = \{E_1, E_2, E_3\}$  are for the three representations and each embedding is a fixed length continuous-valued vectors. For each fusion representation  $FR$ , a tuple  $t = h_1, h_2, h_3$  is generated, where  $h_i = True$  means  $FR$  contains this representation and  $h_i = False$  otherwise.  $t$  can assume 8 possible values, i.e.,  $t = \{FFF, FFT, \dots, TTT\}$ . Among them,  $FFF$  means that none of the three representations are selected, which make no sense for answering RQ1, and thus  $FFF$  is eliminated. These  $FR$  sets are taken as input to and evaluate the precision, recall and F1-measure.

**Table 3: Precision (P), recall (R) and F1 of all representations**

ID	Text	AST	CFG	P	R	F1
1	F	F	T	0.82	0.92	0.87
2	F	T	F	0.91	0.62	0.74
3	<b>F</b>	<b>T</b>	<b>T</b>	<b>0.97</b>	<b>0.95</b>	<b>0.96</b>
4	T	F	F	<b>0.99</b>	0.21	0.35
5	T	F	T	0.85	0.52	0.65
6	T	T	F	0.94	0.44	0.60
7	T	T	T	0.98	0.92	0.95

Table 3 shows the precision, recall, and F1-measure of seven combinations of code representations. It can be observed that  $ID = 3(FTT)$ , the fusion of AST and CFG, achieves the highest F1 values than others, and it validates that our proposed code representation can capture the semantics of the source code for functional clone detection. For  $ID = 1(FFT)$ , the recall of this representation is pretty high. It is because the source codes in the dataset have relatively simple control flow, and many functionality pairs belonging to different problems are mistaken for code clones. For  $ID = 2(FTF)$ , the precision of it is high but has a relatively low recall. It is due to different students may realize the same functionality with different control flow, and this fusion cannot recognize this difference. Similarly, it can be observed that the same phenomenon as  $ID = 4(TFF)$ , which also has high precision and low recall. However, the recall of it is much lower than  $ID = 2(FTF)$  because it is not appropriate to regard the source code as a natural language without any processing. Since the source codes submitted by these students tend to use the similar variable names, such as  $a$ ,  $b$  and  $c$ , the similarity between the text is pretty high, which may lead to many mistakes for code clone. For  $ID = 5(TFT)$  and  $6(TTF)$ , after adding the text representation, their recall have improved to a certain extent compared to  $ID = 1(FFT)$  and  $2(FTF)$ , but precision are sacrificed a lot for the same reason as  $ID = 4(TFF)$ . In  $ID = 7(TTT)$ , we use all three representations. Intuitively, it is believed that the more kinds of code representations are used, the more syntactic and semantic information can be extracted, and the higher precision and recall is. However, the experiment result shows that too much information and too strict clone detection requirements greatly reduce the recall,

and improve a little bit of precision. In summary, we choose to use a fusion of AST and CFG representations to conduct clone detection.

#### 4.4 The Effectiveness of Word Embedding and Graph Embedding Techniques

*RQ3: How does our chosen embedding techniques perform in functional clone detection?* In the previous section, we have validated the effectiveness of our proposed code representations to conduct functional code clones. As we all know, there are several famous word embedding techniques: Word2vec and GloVe and graph embedding techniques: Graph2vec, HOPE, SDNE, and Node2vec. In this section, we shall validate the effectiveness of the word embedding and graph embedding techniques we choose for the embedding learning of the fusion of AST and CFG. For the purpose, the identifier sequence extracted from AST and the structure of CFG is taken as the input of these word embedding and graph embedding techniques to get the embeddings of them. Then we take the fusion of the two embeddings as the input of our deep fusion learning model and evaluate the precision, recall, and F1 measure under OJClone.

**Table 4: Precision, recall and F1 of different embedding techniques**

Embedding techniques	P	R	F1
<b>Word2vec+Graph2vec</b>	<b>0.97</b>	<b>0.95</b>	<b>0.96</b>
Word2vec+HOPE	0.88	0.79	0.84
Word2vec+SDNE	0.75	0.56	0.64
Word2vec+Node2vec	0.55	0.79	0.65
GloVe+Graph2vec	0.89	0.81	0.85
GloVe+HOPE	0.84	0.73	0.78
GloVe+SDNE	0.73	0.52	0.61
GloVe+Node2vec	0.54	0.87	0.67

Table 4 shows the precision, recall, and F1 measure of the combinations of two word embedding techniques and three graph embedding techniques. As shown in Table 4, for two kinds of word embedding techniques, Word2vec achieves better performance than GloVe. Compared with Word2vec, Glove adopted some overall statistics to make up for the shortcomings of the co-occurrence model, while the effect was not necessarily better than simple Word2vec in practice [16]. For the first three combinations, the combination of Word2vec and Graph2vec achieves the highest F1 value. That is because different from other approaches, the vectors generated by Graph2vec can reflect the overall graphs. The vectors generated by other approaches can only reflect the node structure, and the average value of all the nodes may lack the structural feature of the graph.

**Sensitivity to the length of embeddings.** In previous experimental settings, we use continuous-valued vectors of length 16 for word embeddings and graph embeddings. In this section, we study the influence of the different lengths of embeddings on functional clone detection performance of our approach, measured by F1 values. Table 5 shows the F1 value tends to be stable at around 0.96 after the length of word embeddings and graph embeddings are more than 16. Thus, we choose 16 as the length of the word embeddings and graph embeddings used in our approaches.

**Table 5: The F1 values for different length of embeddings**

word \ graph	8	16	32	64
word	0.63	0.71	0.76	0.76
8	0.79	<b>0.96</b>	0.93	0.95
16	0.81	0.95	0.96	0.94
32	0.83	0.96	0.96	0.95
64				

## 4.5 The Effectiveness of Machine Learning Techniques

RQ4: How does DNN perform in functional code clone detection comparing to other traditional machine learning techniques? Table 6 reports the performance of SVM, logistic regression, and our approach on OJClone. Compared with the traditional machine learning techniques, our approach can better capture the hidden features from syntactic and semantic vectors. Thus, our approach is efficient for functional code clone detection.

**Table 6: Comparison with machine learning techniques**

Techniques	P	R	F1
SVM	0.62	0.81	0.70
logistic regression	0.66	0.71	0.68
<b>Our approach</b>	<b>0.97</b>	<b>0.95</b>	<b>0.96</b>

## 4.6 Threats to Validity

There are two main threats to the validity. First, only OJClone dataset is used to demonstrate the effectiveness of our approach. However, OJClone is widely used to evaluate code clone detection approaches, such as Deepsim and ASTNN. We also try to turn to the BigCloneBench dataset, which is a widely-used benchmark for code clone detection. However, it is not proper for evaluating our approach, because the dataset does not contain inter-procedural programs while our approach aims to use caller-callee relation to detect inter-procedural code clones. Besides, our current experiment is on C++ programs, and we plan to involve other language in the future.

Second, as incorporating deep learning, the effectiveness of our approach is limited by the quality of the training set. Even we generated tens of thousands clone pairs and tried to deal with data imbalance. As we have release all the data in our studies, we also plan to build a large and representative dataset.

## 5 RELATED WORK

In code clone detection, the representation of source code determines the upper limit of source code information extraction, and it will determine the model design and affects the final performance.

According to different aspects of the use of source code, the representation method can be divided into four categories: text-based, token-based, syntax-based, and semantics-based techniques. Text-based techniques treat source code as text encoding, and token-based techniques parse sources code into tokens, which are then

organized into token sequences. These two code representations are usually used to detect Type-1 and Type-2 code clones. Since our proposed approach is mainly dealing with functional code clone detection, which is regarded as Type-4 code clones, we do not focus on text-based and token-based techniques.

Syntax-based approaches are not sensitive to the order of source code and can detect Type-1, Type-2, Type-3, and partial Type-4 code clones. Syntax-based techniques take syntax rules of the source code into consideration and mainly contain two types: tree-based and indicators-based approaches. Deckard [9] uses syntax parser to parse source code into a syntax tree, then embeds syntax tree into features, finally uses a locally sensitive hash algorithm for clustering to find the similar code. Daniel Perez et al. [27] generates the AST of source code and uses a feed-forward neural network to classify to code clones. Different from these syntax-based approaches, we visit each node in AST by preorder and use word embedding techniques to learn the AST features.

Semantics-based techniques consider the semantic information of source code. Semantic information refers to the information that can reflect the functionality of code snippets. The most commonly used semantic representation is PDG, which is the combination of control dependence and data dependence. The approach proposed by Komondoor [14] uses program slices to find isomorphic subgraphs of PDGs. Sheneamer et al. [31] extract features from abstract syntax trees and program dependency graphs to represent a pair of code fragments to detect functional code clones. However, the shortcoming of PDG-based approaches is high time cost due to the complexity of PDG. Since the operational semantics of CFG is equivalent to that of deterministic PDG [8], we simply use CFG to represent the semantic features.

In addition to the four categories mentioned above, there are also some hybrid approaches for the functional code clone detection. For instance, CDLH [38] learns hash codes by exploiting the lexical and syntactic information for fast computation of functional similarity between code fragments. DeepSim [42], one of the state-of-the-art approaches, encodes both code control flow and data flow into a semantic matrix and uses a deep learning method to detect functional clones based on the semantic matrix. The semantic representation in DeepSim contains syntactic information at the granularity of basic blocks. White et al. [40] fuse information on structure and identifiers from CFG, identifiers and bytecodes, and adopt recurrent neural networks to learn features based on syntactic and semantic analysis. Different from these approaches, we use word embedding to extract AST and graph embedding to extract CFG, then we use the combination of syntactic and semantic features to train a deep learning model.

Tufano et al. [35] uses four different code representation (i.e., identifiers, AST, bytecode, and CFG) for clone clone detection. It uses four code representations to identify the similarity of code pairs separately and calculate the average as the final similarity result. The thought of using AST as syntax representation and CFG as semantics representation is parallel with ours and is very related. Different from Tufano et al., we combine the syntactic and semantic features before using the deep learning model instead of calculating the average. Since considering syntax and semantics, separately may lead to an opposite result, calculating the average directly may not be suitable. Besides, we use Graph2vec instead of other graph

embedding techniques for feature representation. Different from HOPE generating vectors for each node, Graph2vec generates a vector for the overall graph. Since calculating the average of node vectors may hide the related among nodes, using Graph2vec to generate a graph vector in more suitable. Furthermore, we extract the call graph to capture the connected methods, and we combine the related methods as a functionality. For each functionality, we analyze the fusion of their syntax and semantics for functional code clone detection.

## 6 CONCLUSION

We have presented a novel approach to detect functional code clones with code representation generated from the fusion embedding learning of syntactic and semantic information at functionality granularity, and a deep feature learning model that learns the syntactic and semantic features which convert clone detection into a binary classification problem. We have conducted extensive experiments on a large real-world dataset. The results show that our approach achieves a significant advance over state-of-the-art approaches in terms of F1 measure, and it has good detection efficiency after the model training.

## ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for insightful comments. This work is supported partially by National Natural Science Foundation of China(61932012, 61802171), and Fundamental Research Funds for the Central Universities(14380021).

## REFERENCES

- [1] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*. IEEE, 86–95.
- [2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. 2000. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, 98–107.
- [3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591.
- [4] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code compaction of matching single-entry multiple-exit regions. In *Proceedings of the 10th International Static Analysis Symposium*. Springer, 401–417.
- [5] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 855–864.
- [6] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*. IEEE, 117–125.
- [7] Chencheng Hou, Feiping Nie, Xuelong Li, Dongyun Yi, and Yi Wu. 2014. Joint embedding learning and sparse regression: A framework for unsupervised feature selection. *IEEE Transactions on Cybernetics* 44, 6 (2014), 793–804.
- [8] Sohei Ito. 2018. Semantic equivalence of the control flow graph and the program dependence graph. *arXiv preprint arXiv:1803.02976* (2018).
- [9] Lingxiao Jiang, Ghassan Mishergi, Zhendong Su, and Stephane Glondou. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, 96–105.
- [10] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*. ACM, 55–64.
- [11] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. 2014. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188* (2014).
- [12] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.
- [13] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [14] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*. Springer, 40–56.
- [15] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of 8th Working Conference on Reverse Engineering*. IEEE, 301–309.
- [16] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics* 3 (2015), 211–225.
- [17] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. USENIX, 289–302.
- [18] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 872–881.
- [19] Xing Liu and P Gonçalves. 1987. Program translation by manipulating abstract syntax trees. In *Proceedings of the C++ Workshop*. 345–360.
- [20] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S McKinley. 2015. Does automated refactoring obviate systematic editing?. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 392–402.
- [21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
- [22] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.
- [23] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005* (2017).
- [24] Manziba Akanda Nishi and Kostadin Damevski. 2018. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software* 137 (2018), 130–142.
- [25] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1105–1114.
- [26] J-F Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. 1999. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension*. IEEE, 49–56.
- [27] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, 518–528.
- [28] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.
- [29] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*. IEEE, 172–181.
- [30] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 1157–1168.
- [31] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic clone detection using machine learning. In *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*. IEEE, 1024–1028.
- [32] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospischil. 1997. Introduction to multi-layer feed-forward neural networks. *Chemometrics and Intelligent Laboratory Systems* 39, 1 (1997), 43–62.
- [33] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090.
- [34] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 60–70.
- [35] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 542–553.
- [36] Tim A Wagner, Vance Maverick, Susan L Graham, and Michael A Harrison. 1994. Accurate static estimators for program optimization. *ACM Sigplan Notices* 29, 6 (1994), 85–96.
- [37] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1225–1234.

- [38] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. 3034–3040.
- [39] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of 31st IEEE International Conference on Software Engineering*. IEEE, 364–374.
- [40] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.
- [41] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 783–794.
- [42] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 141–151.

# Learning to Detect Table Clones in Spreadsheets

Yakun Zhang

State Key Lab of Computer Sciences,  
Institute of Software, Chinese  
Academy of Sciences  
University of Chinese Academy of  
Sciences  
Beijing, China  
zhangyakun18@otcaix.icas.ac.cn

Zhiyong Zhou

National Engineering Research  
Center of Fundamental Software,  
Institute of Software, Chinese  
Academy of Sciences  
University of Chinese Academy of  
Sciences  
Beijing, China  
zhiyong@nfs.icas.ac.cn

Wensheng Dou

Jiaxin Zhu

State Key Lab of Computer Sciences,  
Institute of Software, Chinese  
Academy of Sciences  
University of Chinese Academy of  
Sciences  
Beijing, China  
{wsdou,zhujiexin}@otcaix.icas.ac.cn

Jun Wei

Dan Ye

State Key Lab of Computer Sciences,  
Institute of Software, Chinese  
Academy of Sciences  
University of Chinese Academy of  
Sciences  
Beijing, China  
{wj,yedan}@otcaix.icas.ac.cn

Liang Xu

Jinling Institute of Technology  
Nanjing, China  
xuliang@jit.edu.cn

Bo Yang

North China University of  
Technology  
Beijing, China  
yangbo090313@163.com

## ABSTRACT

In order to speed up spreadsheet development productivity, end users can create a spreadsheet table by copying and modifying an existing one. These two tables share the similar computational semantics, and form a table clone. End users may modify the tables in a table clone, e.g., adding new rows and deleting columns, thus introducing structure changes into the table clone. Our empirical study on real-world spreadsheets shows that about 58.5% of table clones involve structure changes. However, existing table clone detection approaches in spreadsheets can only detect table clones with the same structures. Therefore, many table clones with structure changes cannot be detected.

We observe that, although the tables in a table clone may be modified, they usually share the similar structures and formats, e.g., headers, formulas and background colors. Based on this observation, we propose *LTC* (*Learning to detect Table Clones*), to automatically detect table clones with or without structure changes. LTC utilizes the structure and format information from labeled table clones and non table clones to train a binary classifier. LTC first identifies tables in spreadsheets, and then uses the trained binary classifier to judge whether every two tables can form a table clone. Our experiments on real-world spreadsheets from the EUSES and Enron corpora show that, LTC can achieve a precision of 97.8% and recall

of 92.1% in table clone detection, significantly outperforming the state-of-the-art technique (a precision of 37.5% and recall of 11.1%).

## CCS CONCEPTS

• Applied computing → Spreadsheets; • Software and its engineering → Software testing and debugging.

## KEYWORDS

Spreadsheet, table clone, structure, format

## ACM Reference Format:

Yakun Zhang, Wensheng Dou, Jiaxin Zhu, Liang Xu, Zhiyong Zhou, Jun Wei, Dan Ye, and Bo Yang. 2020. Learning to Detect Table Clones in Spreadsheets. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3395363.3397384>

## 1 INTRODUCTION

Spreadsheet systems are one of the most successful end-user programming platforms, and have been widely used in various business tasks, including data storage, data analysis, financial reporting and so on [44]. Scaffidi [48] estimated that, in 2012, over 55 million users in the United States worked with spreadsheets. There must be many more users working with spreadsheets nowadays.

Spreadsheets are code, too. Spreadsheets usually play a similar role to source code in conventional programming languages [25]. Similar to code reuse, end users often reuse existing spreadsheets to speed up their development productivity. For example, a user can prepare a new financial report by copying-pasting-modifying an existing one, thus saving amount of time. For two tables created by copy-paste-modify, they share the same or similar computational semantics. We refer to them as a table clone. For example, the two tables in Figure 1a and Figure 1b form a table clone.

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA  
© 2020 Association for Computing Machinery.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3397384>

Similar to code clones in conventional programs [13, 32, 36, 43], table clones can be applied on some important spreadsheet analysis scenarios. First, table clones can be used to find the same errors scattered in many spreadsheets. Second, inconsistent modifications may introduce errors into table clones [18, 30], thus causing financial losses. Table clones can facilitate to detect and fix spreadsheet errors by analyzing their inconsistencies [18]. Third, table clones are usually used to perform the same business task. Thus, data analysis tools, e.g., PowerBI [4], can extract and analyze all table clones together. Fourth, table clones usually share common computational semantics and structures. Extracting table templates from table clones can facilitate end users to create new table instances.

However, there are no records in the spreadsheet systems (e.g., Microsoft Excel) documenting that two tables were created by copy-paste-modify [31]. Therefore, it is important to effectively detect table clones in spreadsheets. Existing clone detection approaches in spreadsheets [18, 30] mainly focus on the clones with the same data or structures. For example, Hermans et al. [30] consider two blocks of numerical cells with (almost) the same values as a data clone. TableCheck [18] considers two blocks of numerical cells with the same headers, i.e., structures, as a table clone. However, our empirical study on real-world spreadsheets from the EUSES [24] and Enron [26] corpora (Section 4.1) shows that 58.5% of table clones involve structure changes, which cannot be detected by existing approaches. Moreover, code clone detection approaches in conventional programs, e.g., CCFinder [36], Deckard [32], CloneDetective [35], and CCLeaner [42], cannot be applied on spreadsheets, because the programming model in spreadsheets is totally different from those in conventional programs.

In this paper, we propose *LTC* (Learning to detect Table Clones), to detect table clones with or without structure changes in spreadsheets. We observe that the structures and formats, e.g., headers, formulas, and cell fonts, can effectively characterize the computational semantics of a table. If two tables share the same or similar structures and formats, they are likely to be a table clone, and share the similar computational semantics. Therefore, we can treat table clone detection as classifying two tables as a clone or non clone based on their structure and format similarities. Specifically, given two tables, we extract 12 features about structures and formats from each table, and compute the similarity score for each feature. Then, we apply supervised classification algorithms on labeled table clones and non table clones to train a binary classifier. With the trained classifier, LTC first identifies tables in spreadsheets, and then compares every two tables to detect table clones.

We evaluate LTC on the real-world spreadsheets from the EUSES [24] and Enron [26] corpora, which are two of the most widely-used corpora for spreadsheet research [7, 19, 22]. The experimental results show that LTC can detect table clones effectively, with a precision of 97.8% and recall of 92.1%. As a comparison, TableCheck [18] can only detect table clones with a precision of 37.5% and recall of 11.1%. This result shows that LTC can significantly outperform existing approaches.

In summary, we make the following contributions in this paper.

- We propose a commonly-used notation in spreadsheets, table clone, in which two tables share the similar computational semantics.

	A	B	C	D	E	F
1	<b>January</b>	<b>Week 1</b>	<b>Week 2</b>	<b>Week 3</b>	<b>Total Hours</b>	<b>Overtime Hours</b>
2	<i>Green</i>	10	10	5	=SUM(B2:D2)	=MAX(E2-120,0)
3	<i>Jones</i>	15	18	20	=SUM(B3:D3)	=MAX(E3-120,0)
4	<i>Smith</i>	13	21	16	=SUM(B4:D4)	=MAX(E4-120,0)
5	<i>Johnson</i>	42	38	43	=SUM(B5:D5)	=MAX(E5-120,0)
6	<i>White</i>	44	40	42	=SUM(B6:D6)	=MAX(E6-120,0)
7	<i>Edwards</i>	23	35	38	=SUM(B7:D7)	=MAX(E7-120,0)

	A	B	C	D	E	F
1	<b>February</b>	<b>Week 1</b>	<b>Week 2</b>	<b>Week 3</b>	<b>Total Hours</b>	<b>Overtime Hours</b>
2	<i>Green</i>	10	10	5	=SUM(B2:D2)	=MAX(E2-120,0)
3	<i>Jones</i>	15	18	20	=SUM(B3:D3)	=MAX(E3-120,0)
4	<i>Smith</i>	13	21	16	=SUM(B4:D4)	=MAX(E4-120,0)
5	<i>Johnson</i>	35	39	30	=SUM(B5:D5)	=MAX(E5-120,0)
6	<i>White</i>	36	43	32	=SUM(B6:D6)	=MAX(E6-120,0)
7	<i>Edwards</i>	30	40		=SUM(B7:C7)	=MAX(E7-80,0)

	A	B	C	D	E	F
1	<b>March</b>	<b>Week 1</b>	<b>Week 2</b>	<b>Week 3</b>	<b>Total Hours</b>	<b>Overtime Hours</b>
2	<i>Green</i>	20	30	22	=SUM(B2:D2)	=MAX(E2-120,0)
3	<i>Jones</i>	17	42	24	=SUM(B3:D3)	=MAX(E3-120,0)
4	<i>Smith</i>	35	25	28	=SUM(B4:D4)	=MAX(E4-120,0)
5	<i>Johnson</i>	30	23	35	=SUM(B5:D5)	=MAX(E5-120,0)
6	<i>Edwards</i>	21	43	30	=SUM(B6:C6)	=MAX(E6-80,0)
7	<i>Amy</i>	30	30	30	90	0

	A	B	C	D	E
1	<b>April</b>	<b>Week 1</b>	<b>Week 2</b>	<b>Total Hours</b>	<b>Overtime Hours</b>
2	<i>Green</i>	10	30	=SUM(B2:C2)	=MAX(D2-80,0)
3	<i>Jones</i>	22	13	=SUM(B3:C3)	=MAX(D3-80,0)
4	<i>Smith</i>	12	20	=SUM(B4:C4)	=MAX(D4-80,0)
5	<i>Johnson</i>	43	38	=SUM(B5:C5)	=MAX(D5-80,0)
6	<i>Amy</i>	44	40	=SUM(B6:C6)	=MAX(D6-80,0)

(a) January

(b) February

(c) March

(d) April

**Figure 1: Four worksheet excerpts extracted from the EUSES corpus [24].** The tables in every two worksheets form a table clone. The cells in the rectangles show the key differences between the current and previous worksheets. The cells marked by a red right-cornered triangle contain errors.

- We propose a learning-based approach, LTC, to detect table clones with or without structure changes, by exploiting the structure and format similarities among tables.
- We implement LTC and evaluate it on real-world spreadsheets from the EUSES and Enron corpora. The experimental results show that LTC can detect table clones effectively.

## 2 TABLE CLONES IN SPREADSHEETS

In this section, we explain table clones and our motivation using an illustrative spreadsheet example.

### 2.1 Motivating Example

Figure 1 shows four worksheet excerpts, which are extracted from the EUSES corpus [24]. These worksheet excerpts are used to analyze the working hours in four months, i.e., from January to April.

In January, Alice created worksheet *January* to analyze the working hours of her team members in January. In February, Alice copied worksheet *January*, and created a new worksheet *February*. She updated the data and formulas of Johnson, White and Edwards. Because Edwards did not work in “Week 3”, Alice left cell D7 in blank, and fixed the formulas in E7 and F7 accordingly. We can see that the computational semantics, structures and formats in worksheet *January* and *February* are almost the same. Similarly, based on worksheet *February*, Alice created worksheet *March*. She removed White (row 6 in Figure 1b) and added a new member Amy (row 7 in Figure 1c). Note that, Alice updated the value in D6, and forgot to update the formulas in E7 and F7, introducing two formula errors. Further, Alice directly filled all values for Amy without using formulas, introducing two missing formula errors in E7 and F7. For worksheet *April*, Alice deleted column “Week 3” and removed Edwards (row 6 in Figure 1c).

## 2.2 Table Clones

Although the four worksheet excerpts in Figure 1 have different data (e.g., January![B5:D7]<sup>1</sup> and February![B5:D7]), and structures (e.g., column B-F in Figure 1c and column B-E in Figure 1d), they have the similar computational semantics, i.e., they are all used to analyze the working hours in the same way. Thus, the tables in every two worksheets in Figure 1 form a table clone. For example, January![A1:F7] and February![A1:F7] form a table clone.

**Definition 1:** A *table* is a rectangular block of cells, prescribing certain business task. For example, January![A1:F7] in Figure 1 form a table for storing and analyzing the working hours in January.

In spreadsheets, table is the key structure for data processing and information presentation [15, 17]. A table typically contains the following four elements. (1) *Header cell*: Header cells describe other cells in a table. For example, January![B1:F1] and January![A1:A7] in Figure 1a are the headers of table January![A1:F7]. Through header cell B1 and A2, we can know that 10 in cell B2 means Green worked for 10 hours in Week 1. (2) *Data cell*: Data cells store the business data, e.g., the numerical cells in January![B2:D7] in Figure 1a. (3) *Formula cell*: Formula cells analyze data in a table, e.g., January![E2:F7] in Figure 1a. (4) *Cell format*: The cell formats can facilitate user inspection. For example, header cells January![B1:F1] are bold and italic, and have bottom borders.

**Definition 2:** A *table clone* is a table pair  $(t_1, t_2)$ , in which table  $t_1$  and  $t_2$  share the same or similar computational semantics, and prescribe the same or similar business task. In Figure 1, the tables in every two worksheets can form a table clone, e.g., table January![A1:F7] and February![A1:F7].

Note that if a table contains only one row/column, it has limited information, e.g., no headers or data. Thus, we require that tables in a table clone have at least two rows and columns. We have several observations on table clones in spreadsheets. First, headers describe the semantics of a table, and tables in a table clone usually share the same or similar headers. For example, table January![A1:F7] and February![A1:F7] share the same headers. Second, formulas present the computations, and table clones usually share the same or similar formulas. Although the formulas in Figure 1a and Figure 1d are

<sup>1</sup>This denotes cells [B5:D7] of worksheet *January* in Figure 1a. We use similar form to reference cells throughout this paper.

A	B	C	D	E	F	G
	Month	Week 1	Week 2	Week 3	Total Hours	Overtime Hours
1						
2 <i>Green</i>	January	10	10	5	=SUM(C2:E2)	=MAX(F2+120,0)
3 <i>Green</i>	Februrary	10	10	5	=SUM(C3:E3)	=MAX(F3+120,0)
4 <i>Green</i>	March	20	30	22	=SUM(C4:E4)	=MAX(F4+120,0)
5 <i>Green</i>	April	10	30		=SUM(C5:D5)	=MAX(F5+80,0)
6 <i>Jones</i>	January	15	18	20	=SUM(C6:E6)	=MAX(F6+120,0)
7 ...	...	...	...	...	...	...
8 <i>Amy</i>	April	44	40		=SUM(C8:D8)	=MAX(F8+80,0)

Figure 2: The summary of four worksheets in Figure 1.

different, they have similar computations, in which they are used to summarize the working hours for each worker. Third, table clones usually share similar cell formats, e.g., fonts, borders. In addition, tables in a table clone usually store different data for similar tasks. For example, the working hours in Figure 1a and Figure 1c are very different. These observations motivate us to detect table clones by learning the similarities of structures and formats in tables.

## 2.3 Potential Applications of Table Clones

Table clones can be further applied on some important spreadsheet analysis scenarios, e.g., error detection, data analysis, and spreadsheet reuse. This motivates us to effectively detect table clones in spreadsheets. We explain these potential applications as follows.

**Detecting errors related to table clones.** In our motivating example, cell E6 and F6 in Figure 1c suffer from formula errors, in which a cell’s formula is wrong. Cell E7 and F7 in Figure 1c suffer from missing formulas, in which a cell is supposed to contain a formula, but it does not. In a table clone, its corresponding cells usually share the same / similar computational semantics. The inconsistencies among the corresponding cells usually indicate errors. For example, cell E7 in Figure 1a and cell E6 in Figure 1c should have the same computational semantics. However, they contain inconsistent formulas. Based on this observation, we can use table clones to detect spreadsheet errors by analyzing their inconsistencies [18, 33]. Further, given a spreadsheet error, e.g., the formula error in cell E6 of Figure 1c, we can extract all related table clones and check whether they suffer from the same error. This also highlights the importance of table clones.

**Data integration and analysis among table clones.** Table clones are usually used to perform the same / similar business task. However, these related tables usually scatter in many worksheets or spreadsheets. For examples, the four tables in Figure 1 scatter in four worksheets. For data integration, one key question is where to find related tables. The detected table clones can be a good source for table integration. Table clones can help manage all related tables, and facilitate integrating all related tables into a consistent form [15]. For example, Figure 2 is the summary of the four tables in Figure 1. This summary table can be easily used by other data analysis tools, e.g., Insights in Excel [3] and PowerBI [4].

**Table template extraction applications.** When end users reuse an existing table, they usually need to delete the original data, and revise the table according to new requirements [51]. These tasks are usually daunting and error-prone. For example, end users need to delete old data, fix formulas and fill new data when creating Figure 1c based on Figure 1b. We cannot extract table templates from only one table. Table clones provide a group of instances about

**Table 1: Table Clone Types**

Variance	Type-1	Type-2	Type-3	Type-4
format	○	○	○	○
data	✗	✓	○	○
formula	✗	✓	○	○
header	✗	✗	✓	○
Row/col insertion	✗	✗	✗	✓
Row/col deletion	✗	✗	✗	✓

Note: ○ = non-exclusion, ✗ = exclusion, ✓ = inclusion

how a table template is used. Given a group of table clones, we can infer how end users reuse and revise these tables, e.g., what semantics and structures should be kept and which cells will be changed. Table clones can be used to extract table templates [8], which can facilitate end users to create new table instances and avoid errors.

## 2.4 Table Clone Types

In conventional programs, code clones are categorized into four different types [13, 42] according to how developers modify code clones. Since spreadsheets have completely different programming model from conventional programs, the tables in a table clone can be modified in different ways. According to how users modify tables in a table clone, we completely redesign the clone categorization, and further categorize table clones into four types. For a spreadsheet table, headers, formulas and data mainly reflect its computational semantics. Therefore, our table clone type categorization reflects how many computational semantics are changed. Table 1 shows the comparison among the four types of table clones. Note that, table clones are exclusive in their types. That's said, a table clone cannot belong to two types in the same time. We describe four types of table clones as follows.

- **Type-1:** Type-1 table clones are identical tables allowing variations in formats, e.g., cell formats (including height, width, border, color, font) and hidden rows / columns.
- **Type-2:** Type-2 table clones have variations on data and formulas, and also allow variations in Type-1.
- **Type-3:** Type-3 table clones have variations on table headers, and also allow variations in Type-2.
- **Type-4:** Type-4 table clones allow extra modifications, e.g., inserting or deleting rows / columns, and also allow variations in Type-3.

We further illustrate the four types of table clones using Figure 1. (1) If we copy table January![A1:F7] into cells [A11:F17] in Figure 1a, we can form a Type-1 table clone, January![A1:F7] and January![A11:F17]. (2) Table January![A1:F7] and February![A1:F7] only have variations in data and formulas, and form a Type-2 table clone. Note that, headers are not changed in these two tables. (3) Table January![A1:F7] and March![A1:F7] have different headers (i.e., row 7), and form a Type-3 table clone. Note that, in Type-3 table clones, two tables have the same size, i.e., row and column numbers. (4) Table January![A1:F7] and April![A1:E6] have different row and column numbers, and form a Type-4 table clone.

## 2.5 Other Clones in Spreadsheets

Hermans et al. [30] consider two blocks of numerical cells with (almost) the same values as a data clone, e.g., January![B2:D4] and February![B2:D4]. Data clones are very different from table clones. TableCheck [18] considers two blocks of numerical cells with the same headers as a table clone, e.g., January![B2:F7] and February![B2:F7]. Ideally, TableCheck can detect Type-1 and Type-2 table clones. Therefore, neither data clone detection nor TableCheck can detect Type-3 and Type-4 table clones. However, our empirical study on real-world spreadsheets (Section 4.1) shows that, 58.5% of table clones belong to Type-3 and Type-4. This motivates us to detect all types of table clones.

## 3 LEARNING-BASED TABLE CLONE DETECTION

The key insight of our approach is that table clones share the similar structures and formats, even though they contain various variations. By using structures and formats, we are able to predict whether two tables can form a table clone.

Figure 3 presents the overview of our approach, LTC. Given some spreadsheets, LTC first identifies tables (Section 3.1), and then extracts 12 features about structures and formats in each table. For every table pair from the identified tables, LTC computes their feature similarities (Section 3.2). Finally, table clones are predicted by leveraging a binary classification on feature similarities. To train a binary classifier, LTC extracts 12 features from labeled table clones and non table clones, and uses Random Forest [14] to obtain a binary classifier (Section 3.3).

### 3.1 Table Identification

A spreadsheet usually contains multiple worksheets. End users may put multiple tables into the same worksheet. For example, Figure 4 shows a worksheet excerpt, which contains two tables, i.e., [A1:G8] and [A10:C14]. It is improper to treat the whole worksheet as a table, since different parts of a worksheet implement different functions, e.g., [A1:G8] and [A10:C14] in Figure 4 have totally different functions. In spreadsheets, we observe that tables are usually circumscribed by empty cells, e.g., empty cells [A9:G9] in Figure 4. We first classify spreadsheet cells into four types, and then identify tables in each worksheet based on cell types.

**3.1.1 Cell Types.** We utilize the approaches proposed by Hermans et al. [29], and classify cells into four types. (a) *Data*: A cell filled with data, e.g., cell B2 in Figure 4. (b) *Formula*: A cell containing a calculation on other cells, e.g., cell E2 in Figure 1a. (c) *Label*: A cell containing text and expressing the meanings of other cells, e.g., cell B1 in Figure 4. (d) *Empty*: A blank cell.

We classify cells in a worksheet as follows. First, we mark all numerical cells without formulas as *data* cells. Second, we mark all numerical cells with formulas as *formula* cells. Third, we mark the remaining cells contain strings as *label* cells. Fourth, we mark all blank cells as *empty* cells.

**3.1.2 Table Identification.** According to Definition 1, a table is a rectangular block of cells, in which related information prescribing certain business task is put together. We observe that tables are usually divided by empty cells and boundaries. For example, in

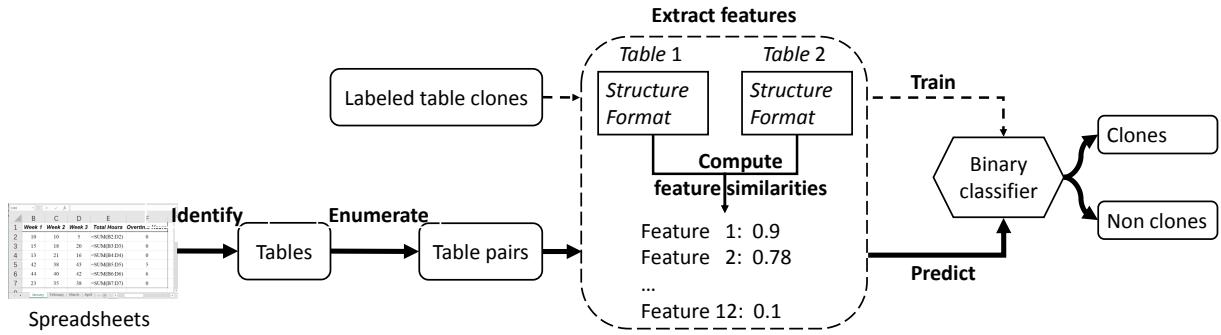


Figure 3: Overview of LTC.

	A	B	C	D	E	F	G
1		Week 1	Week 2	Week 3	Week 4	Total Hours	Overtime Hours
2	Green	10	10.5	5	9	34	0
3	Smith	15	18	20	18	71	0
4	Jones	13	21	16	17	67	0
5	Adams	42	38	43	41	164	4
6							
7	Total hours	80	87.5	84	84	335	4
8	Max hours	42	38	43	41	164	4
9							
10		Pay rate	Gross pay				
11	Green	\$7.50	\$255.00				
12	Smith	\$7.25	\$512.33				
13	Jones	\$8.50	\$566.67				
14	Adams	\$8.25	\$1,353.00				

Figure 4: A worksheet excerpt extracted from the EUSES corpus [24], which contains two tables: [A1:G8] and [A10:C14].

Figure 4, table [A1:G8] and [A10:C14] are divided by empty cells [A9:G9]. Thus, we can use empty cells and boundaries to identify tables in a worksheet.

Given a worksheet, our basic table identification algorithm works as follows. (a) We treat the whole worksheet as a cell block. (b) For a cell block, we identify all empty rows and columns in the cell block. If we find some empty rows or columns, we divide the cell block into multiple cell blocks according to the identified empty rows and columns. (c) For each new identified cell block, we repeat the second step, until we cannot find any new cell blocks.

Take the worksheet in Figure 4 as an example. We first identify two empty rows, i.e., row 6 and 9, and thus divide the worksheet into three cell blocks, i.e., [A1:G5], [A7:G8] and [A10:G14]. Further, column D-G in cell block [A10:G14] are empty, thus, we obtain a new cell block [A10:C14]. So far, we obtain three cell blocks, i.e., [A1:G5], [A7:G8] and [A10:C14].

However, we cannot treat these three cell blocks as three tables. A table can be separated into multiple cell blocks by empty rows and columns. For example, cell block [A1:G5] and [A7:G8] in Figure 4 should belong to the same table [A1:G8], because cells [A7:G8] are used to compute the total and max hours for cells [B2:G5], and empty row 6 is used to separate data and computation. Inspired by ExpCheck [20], we further merge related cell blocks into a big one by using table header information (Section 3.1.3). We observe that, although a table can be separated into multiple cell blocks by empty rows / columns, they usually share the same headers. For example, in Figure 4, cell block [A1:G5] and [A7:G8] share the

Algorithm 1: Identifying column headers in a table.

```

Input: table
Output: headerRows
1 for curRow  $\leftarrow 1$ ;  $curRow \leq 4$ ;  $curRow + +$  do
2   if table.getRow(curRow).isHeader() then
3     | headerRows.add(curRow);
4   else
5     | break;
6 end

```

same column headers (i.e., cells [B1:G1]). Based on this observation, we merge two neighboring cell blocks that share the same row headers or column headers. For the worksheet excerpt in Figure 4, we merge cell block [A1:G5] and [A7:G8] into a new cell block [A1:G8]. Finally, we obtain two tables, i.e., [A1:G8] and [A10:C14].

**3.1.3 Header Identification.** Header cells are used to describe other cells in a table. For example, cells [B1:G1] and [A2:A8] are used to describe data cells [B2:G8] in Figure 4. We use *row header* to denote the header for a row, and *column header* to denote the header for a column. Take table [A1:G8] in Figure 4 as an example. Cell A2 is the row header of row 2, and indicates that cells [B2:G2] belong to “Green”. Cell B1 is the column header of column B, and indicates that cells [B2:B8] belong to “Week 1”.

We have three observations to identify row headers and column headers in a table. (a) Row headers are usually located in the first few (e.g., 4) columns, and column headers are usually located in the first (e.g., 4) few rows. (b) Row headers in a table usually occupy the whole columns, e.g., column A in table [A1:G8] in Figure 4. Similarly, column headers in a table usually occupy the whole row. (c) Row / column headers are usually *label* cells, e.g., cells [A2:A5] in Figure 4. Note that, the only difference between row headers and column headers is their directions. Thus, for clarity, we only use column headers to explain our header identification algorithm.

Algorithm 1 shows how to identify column headers in a table. It starts from the first row of the table, and then checks whether the row contains at least one *label* cells, and all others cells are *empty* (i.e., *isHeader()*). If yes, the *label* cells in the row can be considered as column headers, and it further checks next row of the table, until

we have checked the first four rows in the table. If no, the row can not be considered headers, and the algorithm returns.

Note that, for a cell block discussed in the previous section, we can identify its column headers, too. If we do not find the column headers for a cell block, we further check its nearest cell block *up-Block* on the top, and identify column headers of cell block *upBlock* as its column headers.

Take the worksheet excerpt in Figure 4 as an example. For cell block [A1:G5], Algorithm 1 identifies cells [B1:G1] as its column headers. For cell block [A7:G8], Algorithm 1 does not find its column headers. So, we further identify the column headers of its nearest cell block on the top, i.e., cell block [A1:G5], and consider cells [B1:G1] as its column headers.

### 3.2 Feature Extraction

We observe that structures and formats can effectively characterize a table, and the clones always have similar structures and formats.

Given a table, LTC extracts 12 features of structures. Among these 12 features, some (e.g., font color) are also used by existing work [39], and some of them are unique to table clone detection, e.g., row headers, column headers and formulas. For each table pair, LTC computes a similarity score for each feature, and then constructs a 12-value similarity vector to characterize whether this table pair is a table clone. This similarity vector is further used in training and prediction for table clone detection in Section 3.3.

**3.2.1 Structure Features.** Headers, formulas and cell types can represent the key structures of a table.

As described earlier, headers are used to describe the contents in other cells in a table. For example in Figure 1a, header cell E1 means that cells [E2:E7] are used to compute the total working hours for each team member. Since column headers and row headers describe different aspects of a table, we separate them into two features.

**Column header (Feature #1):** We use the header identification approach in Section 3.1.3 to identify column headers. We denote the column headers of table  $t$  as a set  $H_t$ , then the similarity score  $sim_{F1}$  for column headers in two tables  $t1$  and  $t2$  is calculated as follows:

$$sim_{F1} = \frac{|H_{t1} \cap H_{t2}|}{|H_{t1} \cup H_{t2}|} \quad (1)$$

According to the formula, when two tables are identical and have the same column headers, the similarity score  $sim_{F1}$  is 1. When two tables are totally different and have no column headers in common, the similarity score  $sim_{F1}$  is 0. Thus, the range of similarity score for column headers is  $[0, 1]$ .

**Row header (Feature #2):** Similar to column headers, we use the header identification approach in Section 3.1.3 to identify row headers, and then compute the similarity of row headers in two tables in the same way as column headers.

**Formula (Feature #3):** Formulas in a table usually represent the analyses of the data. Formulas in the corresponding cells among table clones often have the different expressions in the A1 format<sup>2</sup>,

<sup>2</sup>Spreadsheet systems usually have two built-in formats to represent a cell reference: A1 and R1C1 formats. In the A1 format, a cell at the  $x$ -th column and  $y$ -th row is denoted as  $xy$ , e.g., D2. In the R1C1 format, a cell at  $m$  rows below and  $n$  columns right to the current cell is denoted as  $R[m]C[n]$ .

but the same expression in the R1C1 format. For example in Figure 1a, cells [E2:E7] have different formulas in the A1 format, but have the same formula  $SUM(RC[-3]:RC[-1])$  in the R1C1 format. Therefore, we use formulas in the R1C1 format to represent the computations in a table.

A R1C1 formula can be divided into two parts: operators and input variables. Take the R1C1 formula  $SUM(RC[-3]:RC[-1])$  in Figure 1a as an example. Its operators are “SUM”, and its input variables are  $RC[-3]$ ,  $RC[-2]$  and  $RC[-1]$ . We denote the operators used by all formulas in a table  $t$  as a set  $OP_t$ , and input variables as a set  $Var_t$ . The similarity score  $sim_{F3}$  is calculated as follows:

$$sim_{F3} = \left( \frac{|OP_{t1} \cap OP_{t2}|}{|OP_{t1} \cup OP_{t2}|} + \frac{|Var_{t1} \cap Var_{t2}|}{|Var_{t1} \cup Var_{t2}|} \right) \times \frac{1}{2} \quad (2)$$

Our similarity computation for formulas can obtain high similarity scores for similar formulas. Take formulas in Figure 1a and Figure 1d as an example. The R1C1 formula of table in Figure 1a is  $SUM(RC[-3]:RC[-1])$ . The R1C1 formula of table in Figure 1d is  $SUM(RC[-2]:RC[-1])$ . We can see that they share the similar computation. First, they have the same operator “SUM”. Second, formula  $SUM(RC[-3]:RC[-1])$  has three input variables (i.e.,  $RC[-3]$ ,  $RC[-2]$  and  $RC[-1]$ ), and  $SUM(RC[-2]:RC[-1])$  has two input variables (i.e.,  $RC[-2]$  and  $RC[-1]$ ). Their input variables are similar. According to the above similarity computation formulas, their similarity score is 5/6. We can see that, although they have different formulas in Figure 1a and Figure 1d, their similarity score for formulas is high.

**Cell type (Feature #4):** A table can contain *label*, *formula*, *data* and *empty* cells. These cell types can reflect the content structure of a table. By counting the proportion of the four cell types, we can understand the organizational structure of a table. For table clones, they usually share the similar cell type distributions. We use a key-frequency list of  $\langle key, count \rangle$  to represent the cell type distribution in a table, where *key* is a cell type, and *count* shows the occurrence count of type *key*. We use  $L_t$  to denote the key-frequency list for cell types in a table  $t$ . The similarity score of cell types  $sim_{F4}$  is calculated as follows.

$$sim_{F4} = 1 - \frac{\sum_x |freq(L_{t1}, x) - freq(L_{t2}, x)|}{\sum_x |freq(L_{t1}, x) + freq(L_{t2}, x)|} \quad (3)$$

In this formula,  $|freq(L_{t1}, x) - freq(L_{t2}, x)|$  denotes the frequency difference for a cell type  $x$ , and  $freq(L_{t1}, x) + freq(L_{t2}, x)$  denotes the frequency summarization for a cell type  $x$ . Intuitively, more cells in two tables share with the same cell types, the smaller frequency difference each cell type has, the higher similarity score we can obtain.

**3.2.2 Format Features.** Format information is also the important feature to help table clone detection. We observe that two tables that share the similar formats are likely to be a table clone, for example, bold borders, background colors and font-styles. Inspired by this observation, we leverage various format features in our table clone prediction model. We extract the following eight format features for each cell: **font color (Feature #5)**, **font type (Feature #6)**, **font style (Feature #7)**, e.g., bold and italic, **bottom border (Feature #8)**, **top border (Feature #9)**, **left border (Feature #10)**, **right border (Feature #11)**, **background color (Feature #12)**. Take cell A2 in Figure 1a as an example, its font color is Black, its font

type is Geneva, its font style is bold and italic, and it has right border and top border, etc.

For each format feature, we analyze all cells in a table, and build a key-frequency list of  $\langle \text{key}, \text{count} \rangle$ , where  $\text{key}$  is a value for a feature, and  $\text{count}$  shows the occurrence count of  $\text{key}$ . Then, we use the similarity score calculation of the cell type feature to compute the similarity score of each format feature in two tables, as shown in Formula (3).

Note that, we do not consider a data cell's value as a feature. In spreadsheets, the values in a table have the similar role as the inputs of a function in programming languages. Thus, the values cannot usually reflect the computational semantics of a table. For two tables in a table clone, they usually store different values, e.g., Figure 1a and Figure 1d. Identical values are only associated with Type-1 clones, which can be effectively detected based on our current features. Including the feature of values may be helpful for identifying Type-1 clones. However, it may impact the accuracy for identifying other three types of clones. Therefore, we do not consider a data cell's value as a feature.

**3.2.3 Default Settings for Similarity Scores.** For all 12 features, their ranges of similarity scores are  $[0, 1]$ . We observe that not all tables have these 12 features. We need to set default similarity scores. If both tables do not have certain feature, we set the default similarity score of this feature as  $\theta_{both}$ . If one table does not have certain feature but the other does, we set the default similarity core of this feature as  $\theta_{one}$ . According to our experiments in Section 4.2, when  $\theta_{both} = 0.5$  and  $\theta_{one} = 0$ , LTC can obtain the best performance.

### 3.3 Training and Prediction

**3.3.1 Training.** To train a binary classifier for table clone detection, we need a group of table clones and non table clones as our training data. Since there is not such training data for table clone detection, we sample a group of spreadsheets from the EUSES [24] and Enron [26] corpora, which are the most widely used spreadsheet corpora. We further manually identify all table clones and non table clones in these spreadsheets. More details about training data can be found in Section 4.

For each table clone or non clone, we analyze its tables by using the Apache POI library [2], and extract features of each table. These features reflect the semantic characteristics of tables. For each feature, we compute the similarity score of two tables in a clone or non clone. Thus, we obtain a similarity vector of 12 similarity scores. Our training data can be denoted as a list of  $\langle \text{similarity\_vector}, \text{label} \rangle$ , where  $\text{label}$  is 1 for table clones, and 0 for non table clones.

There are many popular classifiers. We set the chosen classifier as parameter  $\theta_{model}$ . According to our experiments (Section 4.2), when we use Random Forest algorithm [14], LTC can obtain the best performance. We use the open source machine learning library scikit-learn [1] to train Random Forest classifier, under its default setting.

**3.3.2 Prediction.** Given some spreadsheets, LTC first identifies all the tables in it. It further removes tables with only one row / column, since they contain very limited information and are hardly used for table clones. LTC then enumerates all possible table pairs, and uses

the trained binary classifier to determine whether a table pair is a table clone or not.

## 4 EXPERIMENTAL DESIGN

In this study, we address the following research questions.

**RQ1:** How effective is LTC in detecting table clones in spreadsheets?

**RQ2:** How is LTC compared with existing techniques, e.g., TableCheck?

**RQ3:** Is LTC robust on different datasets?

To answer RQ1, we evaluate LTC with the built ground truth and analyze its performance. To answer RQ2, we evaluate LTC and TableCheck with the built ground truth to compare their performance. We further analyze the root causes of their performance difference. To answer RQ3, we evaluate LTC on some larger datasets, e.g., FUSE [10], EUSES [24] and Enron [26] and manually validate the detected table clones.

### 4.1 Dataset Construction

**4.1.1 Experimental Subjects.** We select the EUSES [24] and Enron [26] corpora as our experimental subjects. The EUSES corpus was collected from Internet in 2005, and consists of more than 4,000 real-life spreadsheets of 11 categories, e.g., financial, grades and modeling. Since its creation, it has been used by many spreadsheet researches [7, 18, 19, 30, 52]. The Enron corpus was collected from the Enron email archive within the Enron Corporation [38], and contains more than 15,000 spreadsheets. Note that, the spreadsheets in the Enron corpus were not categorized. These two corpora are the most commonly used public industrial spreadsheet datasets.

Table clones are not documented during spreadsheet development. To extract all table clones in spreadsheets, we need to inspect all table pairs in them. Because we are not the authors of these spreadsheets in EUSES and Enron, it is challenging to label table clones for all their spreadsheets. Therefore, we randomly sample a fixed number of spreadsheets from EUSES and Enron to manually build the ground truth. In our experiments, we finally obtain 100 spreadsheets, in which, 50 spreadsheets are from EUSES and 50 spreadsheets are from Enron.

**4.1.2 Sampling Process.** To facilitate the labelling process, we built an Excel plugin, in which participants can mark a region as a table, and two tables as a clone pair in a visual manner. Participants can further view and edit these labelled tables and clone pairs in the plugin. All labelled results were stored in a spreadsheet, which can be used in further analysis.

We build our ground truth through the following steps. (a) We randomly chose a spreadsheet from the corresponding corpus, i.e., EUSES and Enron. We inspected all the worksheets in the sampled spreadsheet for table clones. (b) EUSES and Enron contain multiple versions of a spreadsheet [22, 50], which have very similar structures and contents. To guarantee the diversity of spreadsheets in the ground truth, we excluded this spreadsheet, if it had almost the same structures and contents with any previously selected one. (c) A spreadsheet may contain multiple worksheets that have the same or similar structures. To guarantee the diversity of table clones, for the worksheets with the same or similar structures, we only randomly kept two of them. We also removed worksheets, which were empty or only contain figures. If no worksheet was left in a

**Table 2: Statistics of the Ground Truth**

Corpus	Category	Spreadsheet	Worksheet	Table clone					Non clone
				Type-1	Type-2	Type-3	Type-4	Total	
EUSES	cs101	1	1	0	0	0	1	1	0
	database	9	18	0	3	2	7	12	6
	financial	12	27	2	84	4	44	134	1,349
	forms3	1	2	0	6	4	4	14	239
	grades	4	7	0	7	0	2	9	9
	homework	4	6	7	11	1	8	27	397
	inventory	3	5	0	5	1	10	16	72
	jackson	3	9	6	9	4	12	31	596
	modeling	12	32	1	21	13	32	67	456
	personal	1	2	0	1	0	0	1	5
<b>Total</b>		<b>50</b>	<b>109</b>	<b>16</b>	<b>147</b>	<b>29</b>	<b>120</b>	<b>312</b>	<b>3,129</b>
Enron		50	107	12	95	53	178	338	7,086
<b>Total</b>		<b>100</b>	<b>216</b>	<b>28</b>	<b>242</b>	<b>82</b>	<b>298</b>	<b>650</b>	<b>10,215</b>

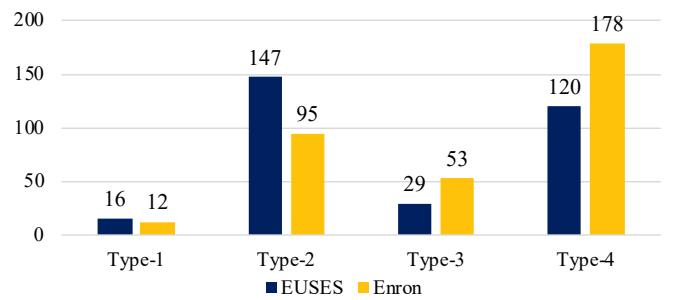
**Table 3: Experimental Design**

	Corpus	Spreadsheet	Table clone					Non clone
			Type-1	Type-2	Type-3	Type-4	Total	
Training	EUSES	35	10	104	8	99	<b>221</b>	2635
	Enron	35	10	57	19	153	<b>239</b>	5487
	<b>Total</b>	<b>70</b>	<b>20</b>	<b>161</b>	<b>27</b>	<b>252</b>	<b>460</b>	<b>8122</b>
Testing	EUSES	15	6	43	21	21	<b>91</b>	
	Enron	15	2	38	34	25	<b>99</b>	
	<b>Total</b>	<b>30</b>	<b>8</b>	<b>81</b>	<b>55</b>	<b>46</b>	<b>190</b>	

spreadsheet, we removed it from consideration. (d) For all remaining worksheets in a spreadsheet, we manually identified all tables in them, and further identified table clones among them. Note that, we not only identify table clones in a worksheet but also among worksheets. For each table clone, we further identified its clone type according to the type categorization in Section 2.4. If a spreadsheet did not contain any table clone, we removed it from consideration. (e) We repeated the above sampling process until 50 spreadsheets for EUSES and Enron were selected, respectively.

Note that it is challenging to manually compare all tables among spreadsheets. Thus, we do not identify table clones among spreadsheets. Meanwhile, having too many similar spreadsheets and worksheets can degrade the performance of our classification model. For example, if a spreadsheet contains 10 similar worksheets and each worksheet contains one table, we can identify  $10^*9/2=45$  table clones of the same family. These table clones can make our model bias to the large table clone families. To address the problem, we removed some spreadsheets and worksheets with the same or similar structures.

To make our sampled spreadsheets and table clones accurate, the first author and two master students carefully inspected the sampled spreadsheets, and tried their best to understand the structures and contents, and further identified tables and table clones. Note that,

**Figure 5: Table clone type distribution in the ground truth.**

we only explained the concepts of table and table clones to the two master students, and did not tell them how LTC works. Finally, they carefully cross-validated all identified tables and table clones.

**4.1.3 Statistics of Ground Truth.** Through the above sampling process, we obtain 100 spreadsheets and identify 650 table clones.

**Table clones:** As shown in Table 2, the spreadsheets in our ground truth are diverse: 50 spreadsheets cover ten categories in EUSES, and 50 spreadsheets come from Enron. We can see that 82

**Table 4: LTC Results on the Ground Truth**

Corpus	Spreadsheet	Table clone	Detected	TP				FP	FN
				Type-1	Type-2	Type-3	Type-4		
EUSES	15	91	86	5	41	19	19	<b>84</b>	2
Enron	15	99	93	2	36	32	21	<b>91</b>	2
<b>Total</b>	<b>30</b>	<b>190</b>	<b>179</b>	<b>7</b>	<b>77</b>	<b>51</b>	<b>40</b>	<b>175</b> (97.8%)	<b>4</b> (2.2%)
									<b>15</b> (7.9%)

(12.6%) and 298 (45.8%) table clones belong to Type-3 and Type-4, respectively. This indicates that table clones with structure changes are common (58.5%). Figure 5 further shows the distribution of table clone types in the ground truth.

**Non table clones:** To train a Random Forest model, we need a set of negative samples, i.e., non table clones. We first manually identify all tables in a spreadsheet in our ground truth. Then, we enumerate table pairs in each spreadsheet. If a table pair is a table clone in our ground truth, we remove it from consideration. All remaining table pairs are non table clones. As shown in Table 2, we obtain 10,215 non table clones.

## 4.2 Experimental Setting

**Training dataset (70 spreadsheets):** We randomly select 35 spreadsheets from EUSES and Enron in the ground truth, respectively. We use all 460 table clones in these 70 spreadsheets as positive samples, and all 8,122 non clones in these 70 spreadsheets as negative samples. The first part in Table 3 shows the statistics of spreadsheets used in the training.

**Testing dataset (30 spreadsheets):** We use the remaining 30 spreadsheets to evaluate LTC. The second part in Table 3 shows the statistics of spreadsheets used in the performance evaluation. We compare LTC with TableCheck on these 30 spreadsheets, too.

**Parameter setting:** In our approach, two default similarity scores  $\theta_{both}$  and  $\theta_{one}$  in Section 3.2.3 should be determined. Their candidate values can be 0, 0.5, and 1. The binary classifier model  $\theta_{model}$  in Section 3.3 should also be determined. The candidate values for  $\theta_{model}$  is Random Forest [14], SVM [34], NuSVC [37], Decision Tree [47] and Logistic Regression [41].

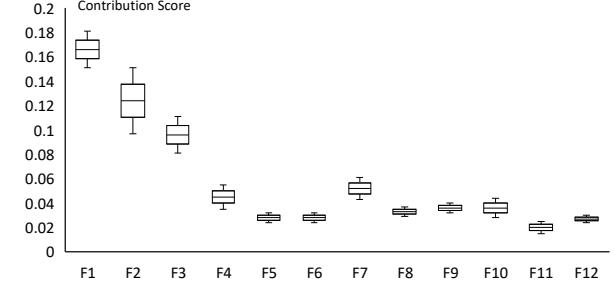
We consider all candidate values of the three parameters, and there are 45 ( $3 \times 3 \times 5$ ) combinations in total. We use 10-fold cross-validation in our training for each combination and calculate the following accuracy indicators: precision, recall and F1-measure. According to the experiment result, we obtain the following best candidates:  $\theta_{both} = 0.5$ ,  $\theta_{one} = 0$ , and  $\theta_{model} = \text{Random Forest}$ .

## 5 EXPERIMENTAL RESULTS

### 5.1 Table Clone Detection Results

We run LTC on the 30 spreadsheets in the testing set of Table 3. Table 4 shows our table clone detection results. It gives the numbers of detected table clones (Detected), and true table clones of each type (TP/Type-1, 2, 3, 4) for the EUSES and Enron spreadsheets.

LTC reports 179 table clones in total, and 175 (97.8%) table clones are true. LTC misses 15 table clones in the ground truth, i.e., the



**Figure 6: Feature analysis. F1 to F12 correspond to the features in Section 3.2.**

recall for table clone detection is 92.1%. Thus, F1-measure of LTC for table clone detection is 94.9%.

**False positives of table clone detection:** LTC wrongly detects 4 table clones (FP in Table 4). We further investigate the causes for these false positives. There are two reasons for these false positives. (1) Our table identification algorithm in Section 3.1 is imprecise. First, end users may put some unrelated data and comments near a table, and our table identification algorithm wrongly considers these data and comments as a part of the table. Second, end users may put multiple tables together, without being separated by empty rows / columns. Thus, our table identification algorithm wrongly considers them as a big table. Imprecise table identification causes three false positives. A more precise table identification approach can help improve LTC's precision. (2) The learned classifier for table clone detection may make wrong predication. In some cases, the table pairs may have some similar features in cell fonts and cell types, and the learned classifier wrongly considers them are table clones. One false positive belongs to this case.

**False negatives of table clone detection:** LTC misses 15 table clones (FN in Table 4). The reasons for the missed table clones are the same as the false positives of table clone detection. First, four false negatives are caused by imprecise table identification algorithm, which does not identify the tables in the ground truth. Second, the learned classifier wrongly considers table clones as non table clones. Eleven false negatives belong to this case.

**Feature analysis:** We use *ReleifAttributeEval* and *Ranker* evaluator provided by Weka [5], to evaluate the importance of 12 features in table clone detection. We adopt 10-fold cross validation approach to do feature analysis in Weka, and the result is shown in Figure 6. We can see that our 12 features all contribute to the table clone detection. According to the contribution of each feature, the rank of

**Table 5: TableCheck Results on the Ground Truth**

Corpus	Spreadsheet	Table clone	Detected	TP				FP	FN	
				Type-1	Type-2	Type-3	Type-4			
EUSES	15	91	30	5	9	0	0	14	77	
Enron	15	99	26	2	5	0	0	7	92	
<b>Total</b>	<b>30</b>	<b>190</b>	<b>56</b>	<b>7</b>	<b>14</b>	<b>0</b>	<b>0</b>	<b>21</b> (37.5%)	<b>35</b> (62.5%)	<b>169</b> (88.9%)

contributions are as follows (from high to low): column header, row header, formula, font bold, cell type, border top, font color, border bottom, border left, background color, border right and font type.

Based on the above analyses, we can draw the following conclusion to RQ1: *LTC is effective in detecting table clones in spreadsheets. The precision and recall for LTC is 97.8% and 92.1%, respectively.*

## 5.2 Comparison with TableCheck

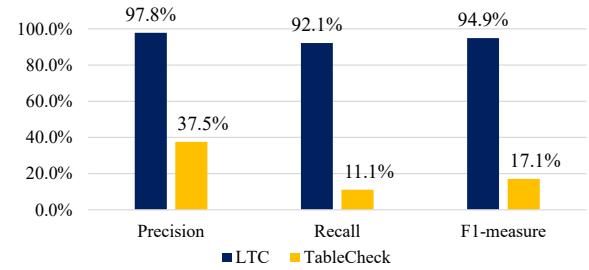
We compare LTC with TableCheck [18] on their table clone detection capability. Note that, we do not compare LTC with data clone detection [30], since a data clone is a pair of two numerical cell blocks with (almost) the same values, which differs from table clones.

TableCheck can only detect table clones with the same headers, e.g., Type-1 and Type-2 table clones, and cannot detect Type-3 and Type-4 table clones. TableCheck extracts table clones by grouping cells with the same headers. First, TableCheck extracts row headers and column headers for each numerical cell. Different from LTC, TableCheck uses the nearest *label* cells as a cell's row or column headers. If a cell does not have row or column headers, TableCheck excludes it from consideration. Second, TableCheck searches for two cell blocks, in which all corresponding cells share the same row and column headers. The *table* definition in TableCheck is different from the common *table* concept used in existing studies [15, 17]. For example, cells January![A1:F5] in Figure 1a are considered as a table. However, they are only a partition of table January![A1:F7].

We run TableCheck on the same 30 spreadsheets in the testing set of Table 3, and compare LTC and TableCheck. Since the tables detected by TableCheck do not contain headers, we append their headers to the corresponding tables for fair comparison.

Table 5 shows the results detected by TableCheck on the 30 spreadsheets. TableCheck detects 56 table clones, and 21 of them are true. Thus, TableCheck achieves a precision of 37.5%, recall of 11.1%, and F1-measure of 17.1%. Figure 7 shows the performance comparison between LTC and TableCheck. As a comparison, LTC achieves a precision of 97.8%, recall of 92.1%, and F1-measure of 94.9%. We further analyze why TableCheck performs worse than LTC on these 30 spreadsheets.

**False positives of TableCheck:** TableCheck wrongly detects 35 table clones (FP in Table 5) on account of the incorrect identification of tables. For example, in Figure 1, January![A1:F7] and March![A1:F7] form a Type-3 table clone. However, TableCheck considers January![A1:F5] and March![A1:F5] as a table clone. Because the authors of TableCheck view these incomplete table clones as true, the reported precision (92.2%) in that paper is higher [18].

**Figure 7: Performance comparison of LTC and TableCheck.**

While, LTC first identifies tables in a spreadsheet, and further checks whether a table pair can form a clone. Tables in the reported clones by LTC are usually complete.

**False negatives of TableCheck:** TableCheck misses 169 table clones (FN in Table 5). TableCheck misses 68 Type-1 and Type-2 table clones. There are two reasons for these false negatives. First, some cells in Type-1 and Type-2 table clones do not have row headers or columns headers, and thus the table clones are excluded by TableCheck. Second, the header identification algorithm in TableCheck does not identify headers correctly, thus missing table clones. TableCheck misses all 101 Type-3 and Type-4 table clones, since TableCheck requires that table clones have the same headers and sizes. LTC utilizes 12 features in spreadsheets, and learning-based similarity comparison approach to avoid these issues faced by TableCheck. Thus, LTC is more universal, and has less restrictions on table structures.

Based on the above analyses, we can draw the following conclusion to RQ2: *LTC significantly outperforms TableCheck in table clone detection. The precision and recall for TableCheck is 37.5% and 11.1%. As a comparison, the precision and recall for LTC is 97.8% and 92.1%, respectively.*

## 5.3 Experiments on Large Datasets

To validate whether LTC is robust on other datasets, we further use our trained model in RQ1 & RQ2 to evaluate LTC on more spreadsheets from FUSE [10], Enron [26] and EUSES [24] corpora.

**Experimental subject:** FUSE [10] is the biggest spreadsheet corpus so far, and contains about 250,000 spreadsheets. In this experiment, we randomly choose 100 spreadsheets from EUSES, Enron and FUSE, respectively.

We apply LTC to detect intra-spreadsheet table clones and inter-spreadsheet table clones on these spreadsheets. Table 6 shows the

**Table 6: Detection Results on Large Datasets**

Spreadsheet			Intra-spreadsheet clone					Inter-spreadsheet clone				
Corpus	Total	Sampled	SS	Clone	Sampled	TP	Precision	SS	Clone	Sampled	TP	Precision
EUSES	4,037	100	65	2,576	100	97	97.0%	43	1,156	100	86	86.0%
Enron	15,926	100	32	289	100	93	93.0%	62	1,332	100	94	94.0%
FUSE	249,376	100	50	1,397	100	98	98.0%	53	2,647	100	97	97.0%
<b>Total</b>	<b>269,339</b>	<b>300</b>	<b>147</b>	<b>4,262</b>	<b>300</b>	<b>288</b>	<b>96.0%</b>	<b>158</b>	<b>5,135</b>	<b>300</b>	<b>277</b>	<b>92.3%</b>

detection result. We detect 4,262 intra-spreadsheet table clones (Intra-spreadsheet clone / Clone), and 5,135 inter-spreadsheet table clones (Inter-spreadsheet clone / Clone) in these spreadsheets. Among these 300 sampled spreadsheets, 147 (49%) spreadsheets contain intra-spreadsheet clones (Intra-spreadsheet clone / SS), and 158 (53%) spreadsheets contain inter-spreadsheet clones (Inter-spreadsheet clone / SS). This also indicates that intra- and inter-spreadsheet table clones are common in practice.

It is challenging to manually validate all detected table clones. Thus, we randomly sample 100 inter-spreadsheet table clones and 100 intra-spreadsheet table clones for each corpus. We follow the same manual inspection procedure described in Section 4.1.2 to check whether they are true table clones. Table 6 shows the validation result. We can see that, LTC can work well on inter- and intra-spreadsheet table clone detection (Precision). LTC can achieve higher (96.0%) precision on intra-spreadsheet table clone detection. The precision of inter-spreadsheet table clone detection is 92.3%, which is also promising. Thus, LTC can also be used to detect inter-spreadsheet table clones.

Based on the above analyses, we can draw the following conclusion to RQ3: *Table clones are common in real-world spreadsheets. LTC can detect inter- and intra-spreadsheet table clones precisely.*

## 6 THREATS TO VALIDITY

Our experiments are subject to several threats to validity.

**Representativeness of experimental subjects.** In our experiment, we select EUSES, Enron and FUSE as our experimental subjects, which have been widely used for many spreadsheet-related studies [6, 18, 19, 25, 30]. We further randomly sample a group of spreadsheets from EUSES and Enron. We believe that our sampled spreadsheets are representative.

**Ground truth and detection result validation.** Since we cannot contact the original authors of the spreadsheets in EUSES and Enron to identify table clones, we have to manually build the ground truth of table clones by ourselves. To alleviate possible mistakes, the first author and two master students carefully cross-validate all table clones in the ground truth and the reported table clones by LTC and TableCheck. In the future, we would like to use crowdsourcing to build larger ground truth to improve LTC further.

## 7 RELATED WORK

Here, we discuss related work that have not been discussed yet.

**Code clone detection.** Code clone detection techniques have been widely studied in conventional programs. There are mainly five types of clone detection techniques [13]. (1) Text-based clone

detection techniques [23] use line-based string matching to detect clones. (2) Token-based clone detection techniques [9, 36, 43] tokenize program code, and use token comparison to detect clones. (3) Tree-based clone detection techniques [12] parse programs into an abstract syntax tree, and then detect clones by tree matching. (4) Graph-based clone detection techniques [40] parse the programs into program dependence graphs, and detect clones as identifying isomorphic subgraphs. (5) Learning-based clone detection techniques [42, 49] adopt machine learning techniques to detect clones. The above code clone detection techniques cannot apply on spreadsheets, since spreadsheets use a different programming model.

**Spreadsheet evolution.** Spreadsheets usually have a long life cycle [27]. Based on spreadsheet filenames’ similarity, VEnron [22] constructs the first versioned spreadsheet corpus, which can be used for spreadsheet evolution studies. SpreadCluster [50] further adopts machine learning to find versioned spreadsheets. However, these corpora and approaches cannot be used for table clone detection in spreadsheets. Our table clone detection approach can be used to facilitate fine-grained spreadsheet reuse analyses.

**Spreadsheet error detection.** Spreadsheets contain various errors [45, 46]. Therefore, many spreadsheet error detection approaches have been proposed. UCheck [7] can detect type inconsistency errors in formulas. Hermans et al. proposed to detect inter-worksheet smells [28], data clone related inconsistencies [30]. AmCheck [19], CACheck [21], CUSTODES [16] and ExcelLint [11] detect errors in similar cells. TableCheck [18] can detect inconsistencies in Type-1 and Type-2 table clones. LTC can detect more table clones (e.g., Type-3 and Type-4) precisely, and makes it possible to detect errors in more types of table clones.

## 8 CONCLUSION

Table clones are widely used to perform similar business tasks in spreadsheets. We observe that the tables in a table clone usually share the similar structures and formats. Based on this observation, we propose a learning-based approach, LTC, to detect table clones with or without structure changes. Our experiments on real-world spreadsheets from the EUSES and Enron corpora show that, LTC can detect table clones effectively, and significantly outperforms existing table clone detection techniques. In the future, we plan to pursue the following research directions. First, we plan to detect inconsistency errors among table clones detected by LTC. Second, we plan to extract table templates based on table clones, and use them in spreadsheet development. Third, we plan to evaluate LTC’s practical effectiveness in companies, and explore new application scenarios of table clones.

## ACKNOWLEDGEMENTS

We thank Shuo Wang and Jiahong Zhou for their contributions in developing the Excel plugin for labelling. This work was partially supported by National Key National Key Research and Development Program of China (2017YFB1001804), National Natural Science Foundation of China (61702490), Microsoft Research Asia Collaborative Research Program, Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), Youth Innovation Promotion Association at Chinese Academy of Sciences, and Beijing College Students' Research Project of High-Level Cross Cultivation of Undergraduate. Wensheng Dou is the corresponding author of this paper.

## REFERENCES

- [1] 2007. *scikit-learn: Machine learning in Python*. Retrieved Jan 15, 2020 from <https://scikit-learn.org>
- [2] 2020. *Apache POI - the Java API for Microsoft Documents*. Retrieved Jan 15, 2020 from <https://poi.apache.org/>
- [3] 2020. *Ideas in Excel*. Retrieved January 15, 2020 from <https://support.office.com/en-ie/article/ideas-in-excel-3223aab8-f543-4fd4-85ed-76bb0295ff4>
- [4] 2020. *Power BI | Interactive Data Visualization BI Tools*. Retrieved Jan 15, 2020 from <https://powerbi.microsoft.com>
- [5] 2020. *Weka 3: Machine Learning Software in Java*. Retrieved Jan 15, 2020 from <http://www.cs.waikato.ac.nz/ml/weka>
- [6] Robin Abraham and Martin Erwig. 2004. Header and unit inference for spreadsheets through spatial analyses. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 165–172.
- [7] Robin Abraham and Martin Erwig. 2007. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages and Computing* 18, 1 (2007), 71–95.
- [8] Robin Abraham, Martin Erwig, Steve Kollmansberger, and Ethan Seifert. 2005. Visual specifications of correct spreadsheets. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 189–196.
- [9] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*. 86–95.
- [10] Titus Barik, Kevin Lubick, Justin Smith, John Slankas, and Emerson Murphy-Hill. 2015. Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets. In *Proceedings of Working Conference on Mining Software Repositories (MSR)*. 486–489.
- [11] Daniel W. Barowy, Emery D. Berger, and Benjamin Zorn. 2018. ExeLint: Automatically finding spreadsheet formula errors. In *Proceedings of International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 148:1–148:26.
- [12] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance (ICSM)*. 368–377.
- [13] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering (TSE)* 33, 9 (2007), 577–591.
- [14] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
- [15] Zhe Chen and Michael Cafarella. 2014. Integrating spreadsheet data via accurate and low-effort extraction. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1126–1135.
- [16] Shing-Chi Cheung, Wanjun Chen, Yeping Liu, and Chang Xu. 2016. CUSTODES: Automatic spreadsheet cell clustering and smell detection using strong and weak features. In *Proceedings of International Conference on Software Engineering (ICSE)*. 464–475.
- [17] Haoyu Dong, Shijie Liu, Shi Han, Zhouyu Fu, and Dongmei Zhang. 2019. TableSense: Spreadsheet table detection with convolutional neural networks. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*. 69–76.
- [18] Wensheng Dou, Shing-Chi Cheung, Chushu Gao, Chang Xu, Liang Xu, and Jun Wei. 2016. Detecting table clones and smells in spreadsheets. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 787–798.
- [19] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. 2014. Is spreadsheet ambiguity harmful? Detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of International Conference on Software Engineering (ICSE)*. 848–858.
- [20] Wensheng Dou, Shi Han, Liang Xu, Dongmei Zhang, and Jun Wei. 2018. Expandable group identification in spreadsheets. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 498–508.
- [21] Wensheng Dou, Chang Xu, S. C. Cheung, and Jun Wei. 2017. CACheck: Detecting and repairing cell arrays in spreadsheets. *IEEE Transactions on software Engineering (TSE)* 43, 3 (2017), 226–251.
- [22] Wensheng Dou, Liang Xu, Shing-Chi Cheung, Chushu Gao, Jun Wei, and Tao Huang. 2016. VEnron: A versioned spreadsheet corpus and related evolution analysis. In *Proceedings of International Conference on Software Engineering (ICSE)*. 162–171.
- [23] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. 2004. Lightweight detection of duplicated code - A language-independent approach. *Institute for Applied Mathematics and Computer Science, University of Berne* (2004).
- [24] Marc Fisher and Gregg Rothermel. 2005. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. 30, 4 (2005), 1–5.
- [25] Felienne Hermans, Bas Jansen, Sohon Roy, Eftimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. 2016. Spreadsheets are code: An overview of software engineering approaches applied to spreadsheets. In *Proceedings of International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 56–65.
- [26] Felienne Hermans and Emerson Murphy-Hill. 2015. Enron's spreadsheets and related emails: A dataset and analysis. In *Proceedings of International Conference on Software Engineering (ICSE)*. Vol. 2. 7–16.
- [27] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2011. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of International Conference on Software Engineering (ICSE)*. 451–460.
- [28] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of International Conference on Software Engineering (ICSE)*. 441–451.
- [29] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. 2010. Automatically extracting class diagrams from spreadsheets. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. 52–75.
- [30] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. 2013. Data clone detection and visualization in spreadsheets. In *Proceedings of International Conference on Software Engineering (ICSE)*. 292–301.
- [31] Felienne Hermans and Tijs van der Storm. 2015. Copy-paste tracking: Fixing spreadsheets without breaking them. In *Proceedings of International Conference on Live Coding (ICLC)*.
- [32] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of International Conference on Software Engineering (ICSE)*. 96–105.
- [33] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. 55–64.
- [34] Thorsten Joachims. 1998. Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of European Conference on Machine Learning (ECML)*. 137–142.
- [35] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2009. CloneDetective - A workbench for clone detection research. In *Proceedings of International Conference on Software Engineering (ICSE)*. 603–606.
- [36] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)* 28, 7 (2002), 654–670.
- [37] Zaheer Ullah Khan, Maqsood Hayat, and Muazzam Ali Khan. 2015. Discrimination of acidic and alkaline enzyme using Chou's pseudo amino acid composition in conjunction with probabilistic neural network model. *Journal of Theoretical Biology* 365 (2015), 197–203.
- [38] Bryan Klimt and Yiming Yang. 2004. The Enron corpus: A new dataset for email classification research. In *Proceedings of European Conference on Machine Learning (ECML)*. 217–226.
- [39] Elvis Koci, Maik Thiele, Óscar Romero Moral, and Wolfgang Lehner. 2016. A machine learning approach for layout inference in spreadsheets. In *Proceedings of International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. 77–88.
- [40] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*. 1095–1350.
- [41] S Lee. 2005. Application of logistic regression model and its validation for landslide susceptibility mapping using GIS and remote sensing data. *International Journal of Remote Sensing* 26, 7 (2005), 1477–1491.
- [42] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CClearner: A deep learning-based clone detection approach. In *Proceedings of International Conference on Software Maintenance and Evolution (ICSME)*. 249–260.
- [43] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering (TSE)* 32, 3 (2006), 176–192.
- [44] Ephraim R McLean, Leon A Kappelman, and John P Thompson. 1993. Converging end-user and corporate computing. *Commun. ACM* 36, 12 (1993), 78–90.
- [45] Raymond R Panko. 2008. Spreadsheet errors: What we know. What we think we can do. *arXiv preprint arXiv:0802.3457* (2008).

- [46] Stephen G. Powell, Kenneth R. Baker, and Barry Lawson. 2008. A critical review of the literature on spreadsheet errors. 46, 1 (2008), 128–138.
- [47] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics* 21, 3 (1991), 660–674.
- [48] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 207–214.
- [49] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 87–98.
- [50] Liang Xu, Wensheng Dou, Chushu Gao, Jie Wang, Jun Wei, Hua Zhong, and Tao Huang. 2017. SpreadCluster: Recovering versioned spreadsheets through similarity-based clustering. In *Proceedings of International Conference on Mining Software Repositories (MSR)*. 158–169.
- [51] Liang Xu, Wensheng Dou, Jiaxin Zhu, Chushu Gao, Jun Wei, and Tao Huang. 2018. How are spreadsheet templates used in practice: A case study on Enron. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 734–738.
- [52] Liang Xu, Shuo Wang, Wensheng Dou, Bo Yang, Chushu Gao, Jun Wei, and Tao Huang. 2018. Detecting faulty empty cells in spreadsheets. In *Proceedings of International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 423–433.

# ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity

Ali Ghanbari

University of Texas at Dallas  
Richardson, TX 75080, USA  
ali.ghanbari@utdallas.edu

## ABSTRACT

In the context of test case based automatic program repair (APR), patches that pass all the test cases but fail to fix the bug are called *overfitted* patches. Currently, patches generated by APR tools get inspected manually by the users to find and adopt genuine fixes. Being a laborious activity hindering widespread adoption of APR, automatic identification of overfitted patches has lately been the topic of active research. This paper presents engineering details of ObjSim: a fully automatic, lightweight similarity-based patch prioritization tool for JVM-based languages. The tool works by comparing the system state at the exit point(s) of patched method before and after patching and prioritizing patches that result in state that is more similar to that of original, unpatched version on passing tests while less similar on failing ones. Our experiments with patches generated by the recent APR tool PraPR for fixable bugs from Defects4J v1.4.0 show that ObjSim prioritizes 16.67% more genuine fixes in top-1 place. A demo video of the tool is located at <https://bit.ly/2K8gnYV>.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Automatic Program Repair, Patch Prioritization, Object Similarity, Test Case

### ACM Reference Format:

Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, July 18–22, 2020, Virtual Event, USA. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404362>

## 1 INTRODUCTION

Manual debugging is notoriously difficult and costly. Automatic program repair (APR) [11] aims to reduce the costs by suggesting high-quality patches that either directly fix the bugs or help the human developers during manual debugging. Generate-and-validate

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

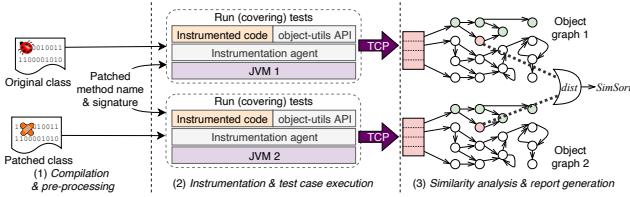
<https://doi.org/10.1145/3395363.3404362>

(G&V) refers to the class of APR techniques that attempt to fix the bug by first generating a pool of patches and validating the patches via certain rules and/or checks. A patch is said to be *plausible* if it passes all the checks. Ideally, we would apply a sound method (e.g., formal verification) for checking validity of generated patches. However, in real-world situations, formal specifications of software are usually absent and due to theoretical limitations, formal verification is in general not automatable. In contrast, testing is the prevalent, economic method of getting more confidence about the quality of software. So, a vast majority of G&V APR techniques, indeed almost all APR techniques, use test cases as correctness criteria for patches [7].

A test-based G&V APR algorithm receives as input a buggy program and, optionally, a test suite consisting of at least one failing test identifying the bug, and produces zero or more plausible patches, i.e., patches that pass all the tests. A typical APR process in this class starts with fault localization to locate likely faulty program locations. It then attempts to fix the bug by applying a number of transformation operators on the identified suspicious locations to obtain a pool of candidate patches to be tested against the existing test suite. Patches that pass all the test cases are reported as plausible patches.

Unfortunately, test cases only partially specify the behavior of the programs and many of the generated patches happen to pass all of the test cases without actually fixing the bug. This makes APR techniques produce many plausible but incorrect patches, aka *test case overfitted* patches [17] (or simply overfitted patches). The process of examining APR-generated patches has to be manual for the oracle problem is undecidable, but in some cases, manually analyzing each and every one of the plausible patches could be even costlier than directly fixing the bug [17]. Thus, a convenient method for post-processing the generated patches before reporting them to the developers is a need. This need is particularly pronounced in the case of APR techniques that are able to explore large search spaces and finding genuine patches that are reported after tens of incorrect ones (e.g., in [10]).

This paper introduces ObjSim, a lightweight, fully automatic patch prioritization tool based on object similarity heuristic. Users of test case based G&V APR techniques designed for JVM-based languages [22] are the envisioned users of this tool. ObjSim prioritizes patches that are more likely to be correct so that the users of APR techniques spend less time examining the generated patches. The tool works by comparing system state at the exit point(s) of the patched method before and after patching and prioritizing patches that result in system state that is more similar to that of original, unpatched version on passing test cases while less similar on failing



**Figure 1: Overall workflow of ObjSim, given class files for original and patched classes, full name of the patched method, and the set of test cases covering the patched method. DFS traversal of object graphs and distance calculator for computing similarity of objects used in SimSort is also depicted.**

tests. The idea is that the behavior of the original program on passing test cases can be used as a partial specification of the desired behavior of the system, so we want a plausible patch to not only not to break passing test cases but also end up in a state similar to that of original version on passing tests. Meanwhile, we want the patch to deviate from original version on failing tests, thereby requiring patches to avoid known erroneous states.

We have applied ObjSim on 358 plausible patches produced by the recent APR tool PraPR for 55 fixable bugs from Defects4J v1.4.0 [6]. We conducted our experiments on a commodity PC and set a time limit of 5 minutes and 16 GB of heap space for each bug. Compared to the original ranking scheme of PraPR, ObjSim prioritizes 5 more genuine fixes in top-1 position (16.67% improvement) and reduces average rank of genuine fixes from 3.04 to 2.74 (almost 10% improvement), yet by relying only on runtime data, it is JVM language agnostic. ObjSim, along with more details about our experiments, is publicly available [8].

## 2 A TOP-DOWN OVERVIEW OF OBJSIM

Virtually all available G&V APR techniques make one-point, or at most one-hunk, changes to the subject programs [7]. Thus, we built ObjSim based on the assumption that patching happens inside a single method. The basic idea of similarity-based patch prioritization is to compare system state at the exit point(s) of the patched method before and after patching and prioritizing patches that result in system state that is more similar to that of original, unpatched version on passing test cases while less similar on failing tests.

Figure 1 depicts an architectural overview of ObjSim and the steps taken for calculating similarity of system state at the exit point(s) of patched method between original and patched versions. ObjSim starts by obtaining class files for original and patched versions of the patched class and full name of the patched method. Some APR tools (e.g., [10]) already provide the needed artifacts/information; in case any of these are absent (e.g., [15] that produces only a source-level patch file), we may obtain them by applying the patch on the corresponding source file, compiling the file, and obtaining the name of the patched method via diffing.

The tool then instruments the original and patched versions of the patched method so that the instrumented program will capture a snapshot of the system state at every exit point of the patched method. This is done with the help of Java Agent technology [16]

and using Javassist library [3] which allows inserting *after advices* in the form of `finally` blocks. The instrumentation code uses Java reflection to capture and serialize the object graphs reachable from all the parameters of the patched method. Then separate processes (i.e., JVM 1 and JVM 2 in Figure 1) are created to execute passing-/failing tests against the instrumented programs while containing side-effects of test execution. Each version of instrumented program is once executed against (covering) originally passing test cases and once against (covering) originally failing tests. Note that restriction to only covering test cases is not necessary and it is done solely for speeding up the entire process. Note also that the two JVM instances can run in parallel to further speed up the prioritization process. Each round of test case execution results in a number of system state snapshots for original and patched version of the instrumented program.

We use  $S(I, m, t)$  to denote the set of snapshots of system state at the exit point(s) of a method  $m$  in the instrumented program  $I$  resulting from executing covering test case  $t$ . Note that  $|S(I, m, t)| \geq 1$ , as a method might be called multiple times. Given a patch  $\pi$  targeting method  $m$  in the program  $I$ , ObjSim obtains  $S(I, m, t)$  and  $S(\pi(I), m, t)$ , where  $\pi(I)$  denotes program  $I$  with patch  $\pi$  applied on it, for all tests  $t$  of the program. Note further that these sets are constructed inside separate processes (namely JVM 1 or JVM 2 in Figure 1), so we have to send them over to the parent process. ObjSim establishes TCP connections between itself and the child processes and transfers the recorded sets by writing them on the socket. It is worth noting that serializing arbitrary Java objects is a non-trivial engineering undertaking and we omit its details here due to space limitations. We encourage the readers to visit the website of the companion library `object-utils` for more information [9].

Except in the case of patching operations that are regarded as *anti-patterns* [18], and are usually avoided by modern APR techniques, patching a program does not change its control flow in significant ways. Thus, the number of times a method gets called tends to be the same before and after patching, i.e.,  $|S(I, m, t)| = |S(\pi(I), m, t)|$  for all tests  $t$ . In case the condition does not hold, ObjSim puts the corresponding patch in a bucket  $W$ , which is to be ranked based on suspiciousness values (e.g., Ochiai suspiciousness) of the patched locations. For a given program  $I$  with the set of test cases  $t_1, \dots, t_n$ , a patch  $\pi \notin W$  targeting method  $m$ , we use  $l(\pi)$  to denote the sequence  $\langle |S(\pi(I), m, t_1)|, \dots, |S(\pi(I), m, t_n)| \rangle$ .

Let  $P$  be the set of all plausible patches generated by the underlying APR tool. Let  $\sim$  denote a binary relation over  $P$  defined as  $\pi_1 \sim \pi_2$  iff  $l(\pi_1) = l(\pi_2)$ , where  $\pi_1, \pi_2 \notin W$ . It is easy to see that  $\sim$  is an equivalence relation which is simply relating patches that have sequence of system state snapshots of the same length, both before and after patching. Let  $Q = P / \sim = \{\pi_\sim \mid \pi \in (P - W)\}$  be the quotient set of  $P$  induced by  $\sim$ , which is simply the set of sets of patches that have the same value for  $l$ .

Having the sets  $W$  and  $Q$ , ObjSim produces the final ranking by concatenating  $m$  non-empty sequences  $\sigma_1, \dots, \sigma_m$  where  $\sigma_i$  is either  $\langle \pi \rangle$  with  $\pi \in W$  or  $SimSort(Q)$  with  $Q \in Q$ .  $SimSort(Q)$  denotes a sequence of patches in  $Q$  that is sorted according to similarity-based criteria. Clearly,  $m = |W| + |Q|$ . The final sequence is sorted in such a way that  $\sigma_i$  precedes  $\sigma_j$  iff  $MaxSusp(\sigma_i) \geq MaxSusp(\sigma_j)$ , where  $MaxSusp(\sigma)$  denotes the maximum suspiciousness value (e.g., Ochiai) for the patched locations corresponding to the patches

in sequence  $\sigma$ . In what follows, we present a more detailed explanation on the algorithm for computing *SimSort*.

## 2.1 Computing *SimSort*

Given a set  $Q$  of patches, *SimSort* returns a sorted sequence of the patches in  $Q$ . Sorting is done by assigning a score to each patch in  $Q$  and putting the patches according to their scores in descending order. In order to present the algorithm in a more precise way, we define distance matrix  $D = [d_{ij}]_{|Q| \times n}$ , where  $n$  is the number of all test cases in the program. Each row in  $D$  corresponds to a patch in  $Q$  and each entry of the matrix represents the *average distance* of system state in the patched program from that of the original program under some test  $t$ . Specifically,  $d_{ij} = \text{avg}\{\text{dist}(S_o, S_p) \mid S_o \in S(I, m, t_j) \wedge S_p \in S(\pi_i(I), m, t_j)\}$  where  $t_j$  is a failing or passing test case. The function *dist* computes distance between two objects. A more detailed description of this function is presented in the subsection that follows.

*SimSort* uses  $D$  to compute scores matrix  $R = [r_{ij}]_{|Q| \times n}$  as follows. For each  $1 \leq j \leq n$ , the function sorts  $j^{\text{th}}$  column of  $D$  in ascending (descending) order if  $t_j$  is a passing (failing) test case.  $r_{ij}$  is the position of  $\pi_i$  in the sorted  $j^{\text{th}}$  column of  $D$ . The function obtains score of each patch  $\pi_i$  by averaging  $i^{\text{th}}$  row of  $R$ . The final result returned by *SimSort* is obtained by sorting the patches based on their scores in reverse order.

**2.1.1 Computing *dist*.** Given two objects  $s_1$  and  $s_2$  (that could be system states),  $\text{dist}(s_1, s_2)$  is computed via DFS traversal of the object graphs reachable from  $s_1$  and  $s_2$  and accumulating distances of “sub-objects” of the objects in a recursive manner. Specifically, *dist* is defined recursively as follows.

- $\text{dist}(s_1, s_2) = 0$  if  $s_1, s_2$  are equal references or equal primitive-typed objects.
- $\text{dist}(s_1, s_2) = 1$  if  $s_1, s_2$  are unequal primitive-typed objects of the same type.
- $\text{dist}(s_1, s_2) = \text{Levenshtein distance between } s_1 \text{ and } s_2$ , if  $s_1, s_2$  are arrays of the same component type.
- $\text{dist}(s_1, s_2) = \sum_{i=1}^n \text{dist}(v(f_i, s_1), v(f_i, s_2))$  if  $s_1, s_2$  are objects of the same type  $\tau$  and  $f_1, \dots, f_n$  are the names of the fields declared or inherited by  $\tau$ . Furthermore,  $v(f, o)$  is defined to be the value of field  $f$  for object  $o$ .
- $\text{dist}(s_1, s_2) = +\infty$  if  $s_1, s_2$  are objects of different types.

For the sake of simplicity in presentation, we have assumed that the object graphs reachable from  $s_1, s_2$  are acyclic. Many engineering details are also omitted. Please see our implementation [9] for more details. The rationale behind the above rules is to extend Levenshtein distance algorithm [21] to handle arbitrary objects: the distance between a primitive-typed value and another of the same type is 1, the distance between arrays is computed as per the conventional Levenshtein distance algorithm, object distances are computed field-wise in a recursive manner, and the distance between objects of different types is defined to be positive infinity.

## 3 OBJSIM USAGE

After checking out ObjSim from [8] and installing it on the local Maven repository, the tool will be available in the form of a Maven plugin. In order to use ObjSim to prioritize plausible patches, the

user needs to add the following snippet under `<plugins>` tag in the POM file of the target project and list fully qualified names of the failing test cases under the designated tag.

```
<plugin>
  <artifactId>objsim</artifactId>
  <groupId>edu.utdallas</groupId>
  <version>1.0-SNAPSHOT</version>
  <configuration>
    <failingTests>
      <!-- list of failing tests -->
    </failingTests>
  </configuration>
</plugin>
```

The tool expects a CSV file, named `input-file.csv`, under the base directory of the project. The input file is intended to contain information about the patches. Each row of this file describes a patch and has to have the following format.

```
Id, Susp , Method , Class-File , Covering-Tests
```

Where `Id` is a unique integer identifier of the patch corresponding to the line, `Susp` is the suspiciousness value for the patch location, `Method` is the fully qualified name of the patched method used during instrumentation, `Class-File` is the name of the compiled class file of patched class, and `Covering-Tests` is the space-separated list of test cases covering the patched location. Test case names should always be of the form `ClassName.MethodName` where `ClassName` is the fully qualified name of the test class. It is worth noting that we have shipped ObjSim with a tool to construct the input CSV file from fix reports generated by PraPR.

After setting up ObjSim, the tool can be invoked via the command `mvn edu.utdallas:objsim:validate`. The output of the tool shall be a sorted list of patch identifiers stored in a text file under the base directory of the target project. For more information and a demo, please see the companion video at <https://bit.ly/2K8gnYV>.

## 4 RELATED WORK

Automated patch classification has lately attracted the attention of APR research community [17, 23–26, 28]. ObjSim is most related to DiffTGen [23] and the technique introduced in [24]. DiffTGen identifies overfitted patches by finding semantic differences between the original, buggy program and its patched version by presenting values of variables and fields to the user and asking them to decide if the demonstrated behavior is reasonable. On the other hand, [24] is fully automatic and works by comparing execution traces between the original and patched programs near the patched method. Although there is a recent study using DiffTGen for classifying patches [27], it is still unclear whether or not asking the users to decide if a behavior is desired (esp., by printing out the intermediate results of computations) is cost-effective. Also, [24] records details about program execution that might be unnecessary when we reason about final results of computations; not to mention that despite discarding information from recorded traces, the implementation still calls for a large amount of computational resources. Unlike DiffTGen, ObjSim automates the process by computing the similarity between system state at the exit point(s) of the patched method in the original program and its patched version. And unlike

[24], it is lightweight yet it does not need to dismiss information about program behavior.

A body of research is also dedicated to techniques for repairing programs while minimizing semantic difference between the original and the patched versions. Chandra et al. [1] introduce a technique for identification of expressions in a buggy program that if replaced with a *good* repair candidate, will solve the bug. A good repair candidate is the one that corrects the failing executions, yet does not break passing tests. This idea forms the basis of the technique for repairing reactive programs by taking a buggy program as a partial specification of the desired behavior and producing high-quality repairs by deviating from it as least as possible [19]. This is related to Qlose [4], a technique for synthesizing fixes for small-scale student programs to pass all the test cases while the difference between execution traces of the original and the patched versions remains minimal. Although these works are related to ObjSim in the basic idea of comparing runtime state of a given patch with that of original version, the goal of two lines of research is fundamentally different. While ObjSim strives for achieving scalability in patch prioritization and applicability to a wide range of APR techniques [2, 15, 20] and programming languages, Qlose and [19] aim to synthesize a patch that is correct by construction and neither scalability nor applicability are concerns.

Pattern-based repair techniques [10, 12, 14] generate patches based on the transformation operators learned from real-world bug fix commits with the goal of generating patches that are more likely to be correct. Anti-patterns [18] refer to the transformation operators that commonly lead to plausible but incorrect patches. Similar pattern-based patch prioritization is employed by [10, 20]. ODS [26] uses source code level features to discriminate correct patches from incorrect ones. Unlike these techniques, ObjSim does not depend on program text, so it is JVM language agnostic and can be used to prioritize patches generated for programs written in languages other than Java, e.g., Kotlin or Scala.

We conclude this section by discussing other techniques. In [25], Yang et al. introduce OPad that automatically filters out overfitted patches introducing regression by generating test cases so as to fuzz test the patched method and identify patches that manifest pre-defined erroneous behavior (e.g., memory leak or crash). Recently, Gao et al. introduce Fix2Fit [5] that follows a similar approach. These techniques are not expected to be effective in case of programs written in managed programming languages [24]. Le et al. [13] show that semantic-based APR techniques also suffer from overfitting.

## 5 CONCLUSIONS

This paper presents ObjSim, a fully automatic, lightweight similarity-based patch prioritization tool for JVM-based languages. It works by comparing the system state at exit point(s) of the patched method between original program and its patched version, and prioritizing patches that result in system state that is more similar to that of original version on passing tests while less similar on failing tests. The key to scalability of ObjSim is to record and compare

computed object graphs rather than complete execution traces. We observed that this technique can be quite effective, resulting in 16.67% improvement compared to default ranking scheme of state-of-the-art PraPR, yet, being semantic-based, the technique is language agnostic.

## ACKNOWLEDGEMENTS

The author thanks ISSTA reviewers for their insightful comments.

## REFERENCES

- [1] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *ICSE*. 121–130.
- [2] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *TSE* (2019).
- [3] Shigeru Chiba. 2000. <https://bit.ly/2UmMuT>. Accessed: April 2020.
- [4] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *CAV*. 383–401.
- [5] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding Program Repair. In *ISSTA*. 8–18.
- [6] Gregory Gay. 2017. <http://bit.ly/2vxSQwR>. Accessed: April 2020.
- [7] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *TSE* (2017), 34–67.
- [8] Ali Ghanbari. 2020. <http://bit.ly/2I3aMBU>. Accessed: April 2020.
- [9] Ali Ghanbari. 2020. <https://bit.ly/2U4SUxt>. Accessed: April 2020.
- [10] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*. 19–30.
- [11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* (2019), 56–57.
- [12] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *SANER*. 213–224.
- [13] Xuan Bach D Le, Ferdinand Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *ESE* (2018), 3007–3033.
- [14] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *ISSTA*. 31–42.
- [15] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *ISSTA*. 441–444.
- [16] Oracle Corporation. 2020. Java Agent. <https://bit.ly/3czmzFV> Accessed June, 2020.
- [17] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*. 532–543.
- [18] Shin H. Tan, Hiroaki Yoshida, Mukul P Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.
- [19] Christian von Essen and Barbara Jobstmann. 2015. Program repair without regret. In *CAV*. 26–50.
- [20] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*. 1–11.
- [21] Wikipedia contributors. 2020. Damerau-Levenshtein distance – Wikipedia, The Free Encyclopedia. <https://bit.ly/2BrMOAj> Accessed June 2020.
- [22] Wikipedia contributors. 2020. List of JVM languages – Wikipedia, The Free Encyclopedia. <https://bit.ly/3714hvf> Accessed June, 2020.
- [23] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*. 226–236.
- [24] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. 789–799.
- [25] Jingyu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *FSE*. 831–841.
- [26] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *arXiv* (2019).
- [27] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *arXiv* (2019).
- [28] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2018. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *ESE* (2018), 33–67.

# Crowdsourced Requirements Generation for Automatic Testing via Knowledge Graph

Chao Guo  
Tieke He\*  
Wei Yuan  
Yue Guo  
Rui Hao

State Key Laboratory for Novel Software Technology, Nanjing University, China

## ABSTRACT

Crowdsourced testing provides an effective way to deal with the problem of Android system fragmentation, as well as the application scenario diversity faced by Android testing. The generation of test requirements is a significant part of crowdsourced testing. However, manually generating crowdsourced testing requirements is tedious, which requires the issuers to have the domain knowledge of the Android application under test. To solve these problems, we have developed a tool named *KARA*, short for Knowledge Graph Aided Crowdsourced Requirements Generation for Android Testing. *KARA* first analyzes the result of automatic testing on the Android application, through which the operation sequences can be obtained. Then, the knowledge graph of the target application is constructed in a manner of pay-as-you-go. Finally, *KARA* utilizes knowledge graph and the automatic testing result to generate crowdsourced testing requirements with domain knowledge. Experiments prove that the test requirements generated by *KARA* are well understandable, and *KARA* can improve the quality of crowdsourced testing. The demo video can be found at <https://youtu.be/kE-dOiekWWM>.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Android GUI Testing, Crowdsourced Requirements, Knowledge Graph

### ACM Reference Format:

Chao Guo, Tieke He, Wei Yuan, Yue Guo, and Rui Hao. 2020. Crowdsourced Requirements Generation for Automatic Testing via Knowledge Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404363>

\*Corresponding author: hetieke@gmail.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404363>

## 1 INTRODUCTION

Android applications occupy most of the mobile application market. Due to the diversity of Android platform brands and the Android system versions, the fragmentation of the Android system is becoming increasingly severe. Various application scenarios and frequent iteration cycles make Android testing difficult. More and more developers solve the dilemma facing Android testing by using automatic testing and crowdsourced testing.

Android automatic testing is a test method which traverses the Android page components and explores the application state by simulating user interaction events and system events. It can detect potential errors or exceptions in the application by generating testing inputs [5]. The errors and exceptions can be called suspicious bugs because it is unclear whether suspicious bugs during the automatic testing process are real bugs. Moreover, the results of Android automatic testing are difficult for programmers or testers to understand. So in most of the cases, it's hard for programmers and testers to acquire effective crowdsourced testing requirements from the test results directly.

Crowdsourced testing refers to the requesters initiate test tasks to workers who are from different regions. The workers complete the tasks and submit reports [2]. Traditionally, the crowdsourced testing task includes the target application and test requirements. The crowd workers follow the instruction in the test requirement to complete the test task. However, manually generating test requirements is tedious for requesters. It requires requesters to gather domain knowledge of the target application.

In order to resolve the problem of requirements generation, we take advantage of the knowledge graph to automatically obtain the domain knowledge for the target application. As a result, we have developed the *KARA* to generate crowdsourced testing requirements automatically. *KARA* first builds a GUI model and a knowledge graph with domain knowledge for the target application. Then, the search algorithm is adopted to obtain the shortest operating path from the application entry interface to the suspicious bug according to the GUI model. At last, *KARA* utilizes the knowledge graph and the shortest operating path to generate test requirements.

In this paper, we mainly make the following contributions.

- We propose the *KARA* tool, which generates crowdsourced testing requirements for Android application automatically.
- We are the first to come up with taking advantage of the knowledge graph to assist in generating descriptions of crowdsourced testing requirements.

## 2 RELATED WORK

Crowdsourced testing for mobile applications is receiving increasing attention. Komarov et al. [4] compared crowdsourced GUI testing with traditional laboratory testing and proved the feasibility of crowdsourced GUI testing. Maria et al. [5] collected application data from the operations of crowdsourced workers on the application, and reproduced the context scenarios of the exception to help developers locate the cause of the exception.

Automated testing is a process that translates human-driven testing behaviours into machine execution. Amalfitano et al. [1] proposed a GUI ripping-based Android automated testing technology AndroidRipper, which uses a depth-first traversal strategy to explore different states of the program operation. It can explore the GUI changes from different startup states with manual assistance. Hao et al. [3] implemented a universal UI automation framework PUMA, which has certain extensibility, allowing testers to generate application state models based on different exploration strategies.

With the advent of sophisticated Natural Language Processing algorithms, knowledge graph has become popular. Anmol Nayak et al. [6] proposed a knowledge graph creation tool, which can extract the test intent from a requirement statement. It can be used for automated generation of test cases from natural (domain) language requirement statements.

## 3 IMPLEMENTATION

The architecture of KARA is shown in Figure 1. KARA first analyzes the results of the Android automated testing on the target application to obtain window transitions. Then, KARA constructs the GUI model for the target application. The Dijkstra algorithm is adopted to calculate the shortest operation sequence from the application starting interface to the suspicious bug interface. In the meanwhile, the static analysis technology of lexical analysis is utilized to get widget information from the source code. Based on the result of lexical analysis and transition information of the GUI model, KARA constructs a knowledge graph. Ultimately, KARA combines the shortest operation sequence with the knowledge graph to generate crowdsourced testing requirements so that crowdsourced workers can reproduce the suspicious bugs of test requirements.

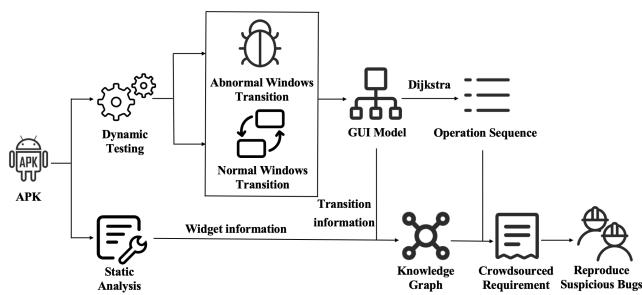


Figure 1: The Architecture of KARA

### 3.1 Operation Sequence Acquisition

In this paper, we use the Android automatic testing tool of Mootest platform<sup>1</sup>, which is assisted by manual test scripts. It uses the input

<sup>1</sup><http://www.mootest.net/>

parameters in the manual test scripts when the input widget is encountered during traversal, which can obtain comprehensive and sufficient test results. According to the test results of the Android automatic testing on the target application, KARA can get the test execution events. KARA constructs the GUI model based on normal and abnormal window transitions information, which is captured during the execution of the events. The Dijkstra algorithm is used to acquire the shortest sequence of events from the application entry interface to the triggered abnormal window transitions interface.

**3.1.1 Normal Windows Transition.** To traverse the events in automated testing process, we use a directed graph  $G = \langle V, E \rangle$  to represent the state changes of the GUI. Vertice set  $V = \langle V_1, V_2, \dots, V_n \rangle$  is an application window list, and edge set  $E = \langle E_1, E_2, \dots, E_n \rangle$  is a window transition list. For  $E_i = \langle V_s, V_t, m, t, c, b, P_e \rangle$ ,  $V_s$  and  $V_t$  represent the starting window and the target window respectively. The operation information of triggering the window transition is recorded in  $m$ .  $t$  represents the time of the event triggered. The classification of window transition event is  $c$ . The default value 1 represents the normal window transition event, 2 represents the abnormal window transition event. The detail of the abnormal event information is recorded in  $b$ . Before the event is triggered, a screenshot would be stored in  $P_e$ . There may be multiple transitions between two windows. Various edges can exist in the same direction between  $V_s$  and  $V_t$ . However, the operation information  $m$  of  $E_x$  are different.

In the process of automated testing, the Android automated testing tool would save screenshots of the Android activity. The screenshot list can be represented as  $P = \langle P_1, P_2, \dots, P_n \rangle$ . For any screenshot  $P_i = \langle n, t \rangle$ , its file name is  $n$ , and the timestamp is  $t$ . In the set of  $E$ , there must exist an edge  $E_j$  whose  $t_j$  is greater than and closest to  $t$ . Then,  $P_i$  is stored in  $P_e$  of  $E_j$ . The code is shown in Algorithm 1 (lines 2-8). KARA establishes a mapping relationship between screenshots and window transitions. It uses screenshots to assist in displaying test steps for crowdsourced requirements.

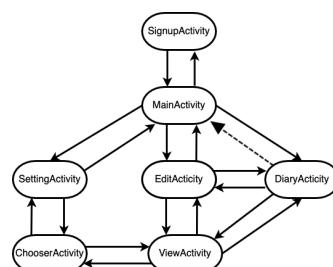


Figure 2: GUI model of the application “JianShi”

**3.1.2 Abnormal Windows Transition.** KARA utilizes the results of automatic testing to establish exception logs and window transitions mapping. The exception log set is  $L = \langle L_1, L_2, \dots, L_n \rangle$ . For any exception log  $L_i = \langle m, t \rangle$ , the timestamp of log formation is  $t$ . In set of  $E$ , there must exist an edge  $E_k$  whose  $t_k$  is smaller than and closest to the  $t$ , thus we can establish the mapping between  $L_i$  and  $E_k$ . The information  $m$  of  $L_i$  is stored in  $b_k$  of  $E_k$ , and  $c_k$  is set to 2 (lines 10-16).

Part of the GUI model of Android application “JianShi” is shown in Figure 2. For the convenience of illustration, An edge is selected in the direction when various edges exist between two nodes. The

solid arrow in the Figure 2 represents the normal window transition, and the dotted arrow represents the abnormal window transition.

**3.1.3 Get the Shortest Operation Sequence.** From the GUI model, the operation sequence  $S = \langle E_1, E_2, \dots, E_n \rangle$  of the application entry interface reaching the abnormal window transition  $E_n$  can be obtained. The  $bfs$  is used to get operation sequence  $S_t = \langle E'_1, E'_2, \dots, E_m \rangle$ , which is the shortest sequence from the application entry interface to the third-to-last event  $E_m$  of  $S$  (line 19–20). Then the last two events  $E_{n-1}, E_n$  are added to  $S_t$ . Then KARA forms the shortest operation sequence  $S' = \langle E'_1, E'_2, \dots, E_m, E_{n-1}, E_n \rangle$  (line 21). Experience shows that the last three steps in the operation sequence can guarantee the correctness of the scene. If the event length of  $S$  is less than three, the shortest operation sequence of the application entry interface to  $E_n$  is obtained directly through the  $bfs$  (line 24).

### Algorithm 1

```

Input:  $G = (V, E), P, L$ 
Output:  $S'$ 
1: Procedure mappingScreenshot(EventSet  $E$ , ScreenshotSet  $P$ ):
2: for all  $P_i \in P$  do
3:   for all  $E_j \in E$  do
4:     if  $E_j.t > P_i.t$  and  $E_j.t \leq P_{i+1}.t$  then
5:        $E_j.P_e \leftarrow P_i$ 
6:     end if
7:   end for
8: end for
9: Procedure mappingBugLog(EventSet  $E$ , bugLogSet  $L$ ):
10: for all  $L_i \in L$  do
11:   for all  $E_k \in E$  do
12:     if  $E_k.t > L_{i-1}.t$  and  $E_k.t \leq L_i.t$  then
13:        $E_k.b \leftarrow L_i, E_k.p \leftarrow 2$ 
14:     end if
15:   end for
16: end for
17: Procedure getShortestPath(SequenceList  $S$ ):
18: if  $S.size() > 3$  then
19:    $S_t = \langle E'_1, E'_2, \dots, E_m \rangle$ 
20:    $S' \leftarrow bfs(S_t)$ 
21:    $S'.add(E_{n-1}), S'.add(E_n)$ 
22:   return  $S'$ 
23: else
24:   return  $bfs(S)$ 
25: end if

```

## 3.2 Knowledge Graph Construction

In the traditional crowdsourced testing, the generation of test requirements requires the requesters gathering domain knowledge of the target application. Fortunately, the knowledge graph can solve the issue of containing domain knowledge. It includes entities, relationships between entities and attributes to represent the ontology. In this paper, the data of knowledge graph construction comes from two parts: the GUI model and the source code of the target application. Based on the GUI model, the transition information can be acquired to form part of the entities and the corresponding relationship in the knowledge graph. The attributes in the knowledge graph can be composed of the attribute information of the widgets from the source code. Therefore, we utilize the GUI model to construct the skeleton structure of a knowledge graph and then use source code to enrich the contents of entities and attributes.

We define the type of entities in two different perspectives. From the interaction view, there are three kind of entities: 1) **Head Entity**: The entity that appeared before an action is performed. 2)

**Tail Entity**: The entity that appears after an action is executed.

3) **Action Entity**: The entity only acts as an action. As shown in Figure 3, The Click Action is executed on **MainActivity**, then the interface is jumped to **EditActivity**. Thus, **MainActivity** is a Head Entity, the Click Action is an Action Entity and **EditActivity** belongs to Tail Entity. From the widget affiliation perspective, another two entity types are defined: 1) **Master Entity**: An entity that owns other entities. 2) **Slave Entity**: An entity that belongs to a certain Master Entity. In the Figure 3, that a button as Slave Entity belongs to **MainActivity** as Master Entity means a button is in the **MainActivity** interface. In our knowledge graph ontology, the relationship between entities can be **BelongTo**, **Take** or **Result**. The **BelongTo** illustrates the relationship between widgets and layouts. The **Take** and **Result** will delineate the cause and effect of actions. Attributes describe the characteristics of entities in a certain situation, which can be defined as the following four types: 1) **ID**: the unique identifier of the entity. 2) **Description**: the description of the operation. 3) **Type**: the value is either Click, Input or Slide. Action Entity node, Master Entity node, Slave Entity node have this attribute. An example of the Knowledge Graph body structure is shown in Figure 3.

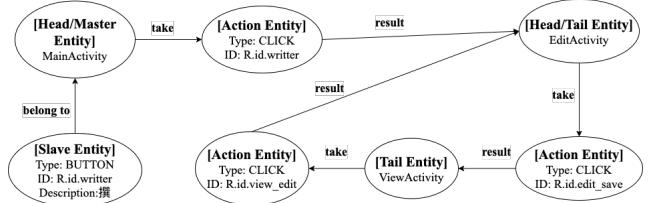


Figure 3: An Example of Knowledge Graph of KARA

## 3.3 Requirement Generation

From the GUI model, KARA can get the shortest operation sequence of actions (like the following code), while it cannot acquire the detailed information of widget, resulted in less informative requirements generation.

```

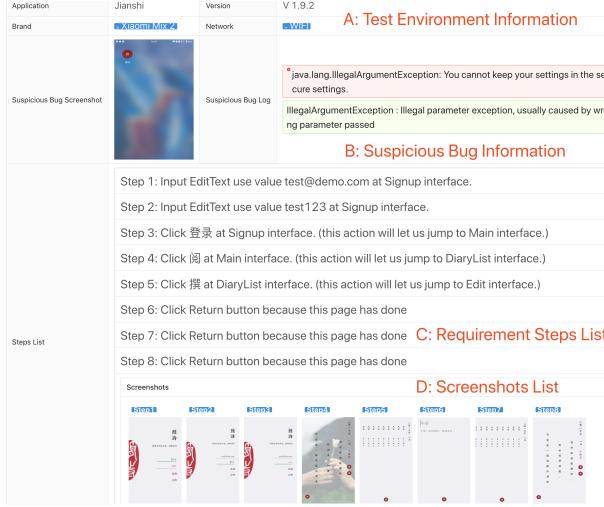
1. Input android.jianshi:id/email use vaule test@demo .
2. Input android.jianshi:id/password use vaule test123 .
3. Click widget android.jianshi:id/login .
4. Click widget android.jianshi:id/reader .
5. Click widget android.jianshi:id/view_write .
6. Click Return button because this page has done .
7. Click Return button because this page has done .
8. Click Return button because this page has done .

```

The knowledge graph is the roadmap of interface interaction meanwhile contains the knowledge of each widget. Based on the shortest operation sequence and the knowledge graph, KARA can generate the crowdsourced requirements. Each event  $E$  of the shortest operation sequence  $S'$  contains event information  $m$ . According to the widget information in  $m$ , the corresponding widget attribute can be found in the knowledge graph. Then, KARA optimizes the widget information description and the window transition information for  $m$  from the knowledge graph. The requirement steps after using the knowledge graph are depicted in Figure 4(PartC).

An example of crowdsourced testing requirements is shown in Figure 4. The requirement includes four parts: Test environment information (Figure 4, PartA), which includes the information of application name, application version, device brand and network

condition. Suspicious bug information (Figure 4, PartB), which consists of the log information and screenshot of the suspicious bug. Requirement steps list (Figure 4, PartC), which contains the operation description of each step. It clearly defines interface transitions and hints for the next transition interface. Screenshots list (Figure 4, PartD), which includes a series of screenshots of the step.



**Figure 4: An example of crowdsourced testing requirement**

## 4 USER STUDY

We conducted experiments to address the following research questions.

**RQ1:** How understandable are the test requirements generated by *KARA*?

**RQ2:** Whether the test requirements generated by *KARA* effectively improve the quality of crowdsourced testing?

To address **RQ1**, we designed a sample and conducted a survey to collect the satisfaction of twenty participants on the comprehensibility of the test requirements generated by *KARA*. The twenty participants are graduate students majoring in software engineering and have been registered on the Mootest platform for more than one year. From the statistical outcomes, we found 90% of participants had a positive attitude towards the understandability of the test requirements. The rest had shown negative attitudes because they thought the description of suspicious bug was insufficient, which affected the judgment on the suspicious bug.

To address **RQ2**, we conducted a controlled experiment. Ten domestic and foreign Android applications were selected from the open-source website, such as *JianShi*, *QQplayer* etc. They have a large number of users. Ten Android devices with high occupancy in the domestic Android market were chosen, such as *HuaWei P30*, *Xiaomi 10Pro* etc. We had installed all the applications before the experiment. Twenty experimenters were divided into two groups whose member tested the applications on one device. Group A is a control group that test by themselves. Group B is an experimental group that test applications according to the test requirements generated by *KARA*. The bug report that contains the bug screenshot and a short description submitted by the experimenters were collected during two hours testing process. After the experiment, the professional testers checked and analyzed the bug reports.

Table 1 shows the evaluation result.  $B_c$  represents the count of bug reports submission in the two hours.  $B_e$  represents the count of valid bugs in the submission. If the applications appear unexpected results such as confusing layouts or unresponsive, we think they are valid bugs. Within two hours, group A and group B submitted 118 and 115 bug reports, of which 68 and 93 bug reports were valid bugs. The proportion of valid bugs reports were 57.6% and 80.9% in total submission bugs, respectively. The result shows that group B testing requirements generated by *KARA* submit fewer bug reports, but it has a high rate with valid bug reports. Therefore, we can find that the test requirements generated by *KARA* can effectively improve the quality of crowdsourced testing.

**Table 1: The evaluation result of test requirements**

App	Version	Type	Group A $B_e/B_c$	Group B $B_e/B_c$
JianShi	2.2.0	Read	7/13	9/11
TesterHome	1.0	Community	9/20	12/15
Leafpic	0.6	Tool	3/8	8/10
RGB Tool	1.4.4	Tool	8/10	9/11
SeeWeather	2.3.1	Tool	6/9	7/9
IThouse	1.0.1	Community	5/13	8/10
QQplayer	3.2	Player	2/6	3/4
GuDong	1.8.0	Sport	5/11	7/11
AnkiDroid	2.8.4	Study	12/15	13/14
SolarCompass	1.0	Location	11/13	17/20

## 5 CONCLUSION

We developed the *KARA* to solve the problem of crowdsourced testing requirements generation. It is tedious to generate requirements manually during the process of crowdsourced testing. *KARA* uses the knowledge graph to enrich the description of the steps in the requirements. It can generate crowdsourced testing requirements automatically. Testers can obtain comprehensive requirements from *KARA*. Moreover, crowdsourced workers can reproduce suspicious bugs easily according to the requirements that generated by *KARA*.

## ACKNOWLEDGEMENTS

This work was partially funded by the National Key Research and Development Program of China (2018YFB1403401) and the Fundamental Research Funds for the Central Universities (14380023).

## REFERENCES

- [1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In ASE. ACM, 258–261.
- [2] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing.. In ESEC/SIGSOFT FSE. 225–236.
- [3] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In MobiSys. ACM, 204–217.
- [4] Steven Komarov, Katharina Reinecke, and Krzysztof Z Gajos. 2013. Crowdsourcing performance evaluations of user interfaces. In SigCHI. 207–216.
- [5] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In ESEC/SIGSOFT FSE. ACM, 224–234.
- [6] Anmol Nayak, Vaibhav Kesri, and Rahul Kumar Dubey. 2020. Knowledge Graph based Automated Generation of Test Cases in Software Engineering. In Proceedings of the 7th ACM IKDD CoDS and 25th COMAD. 289–295.

# TauJud: Test Augmentation of Machine Learning in Judicial Documents

Zichen Guo

Jiawei Liu

Tieke He\*

Zhuoyang Li

Peitian Zhangzhu

State Key Laboratory for Novel Software Technology, Nanjing University, China

## ABSTRACT

The booming of big data makes the adoption of machine learning ubiquitous in the legal field. As we all know, a large amount of test data can better reflect the performance of the model, so the test data must be naturally expanded. In order to solve the high cost problem of labeling data in natural language processing, people in the industry have improved the performance of text classification tasks through simple data amplification techniques. However, the data amplification requirements in the judgment documents are interpretable and logical, as observed from CAIL2018 test data with over 200,000 judicial documents. Therefore, we have designed a test augmentation tool called TauJud specifically for generating more effective test data with uniform distribution over time and location for model evaluation and save time in marking data. The demo can be found at <https://github.com/governormars/TauJud>.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Test Augmentation, Machine Learning, Judicial Documents

### ACM Reference Format:

Zichen Guo, Jiawei Liu, Tieke He, Zhuoyang Li, and Peitian Zhangzhu. 2020. TauJud: Test Augmentation of Machine Learning in Judicial Documents. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404364>

## 1 INTRODUCTION

The development of judicial research in the era of big data has produced many applications[5]. Among those applications, the use of machine learning algorithms combined with referee paper data to predict the case to assist trials has become the current focus[10].

\*Corresponding author: hetieke@gmail.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404364>

Benefited from big data mining, employing machine learning techniques to construct a judicial prediction model is increasingly commonplace in the legal field[4]. These models need to be deployed in the big data environment to process continual evaluations. Not only the algorithms chosen in the judicial prediction model, but also the training set of the judgment documents used as the input affect the performance of the model.

The complexity of machine learning systems makes it difficult for developers to understand the details of system decisions, which has led developers to measure the quality of the system only through test results[6]. Therefore, the development of machine learning systems will rely more on testing than traditional software system development[1]. However, as a critical role in the testing phase, test data does not guarantee its quality, let alone test the quality of intelligent software systems. A high-quality case screening system test set should cover all possible classification results without bias. In contrast, there are specific problems with the existing judicial data, so data augmentation is required for the judgment documents. On the one hand, the larger the scale of the testing data, the higher the accuracy of the prediction model. On the other hand, due to the significant imbalance in the distribution of judicial cases, the data of the judgment documents in some unpopular cases is too small to meet the needs of the machine learning prediction model.

Many researchers have tried many ways to augment the text data[11]. To solve the problem of the high cost of manually labeling data in natural language processing, Jason Wei et al. present a tool called EDA[8]. In order to improve the performance of text classification tasks, simple data augmentation techniques are needed. EDA can help models become more stable and robust with augmentation of testing data. However, it is not applicable to test augmentation in judicial documents that need high interpretability and logic for judgment. It is generally known that a more massive amount of test data can better reflect the real performance of models, so natural augmentation on test data is necessary.

Another popular model called QANet generates new training data by translating sentences into French and back into English[12]. Moreover, synonym replacement is also an ideal way to augment judicial documents. Based on the context of judicial cases, we design particular synonyms of legal terms as corpus. Fairness in machine learning raises concern nowadays[2]. It is an essential requirement of the legal system, where judicial injustice will severely damage the credibility of the relevant departments and organizations. Garg et al [3] proposed blindness and counterfactual augmentation to evaluate the model's fairness.

Accordingly, we design TauJud especially for test augmentation in judicial documents, which can generate more effective test data for model evaluation and save time for labeling data. By combining Universal Augmentation Methods with judicial augmentation, we developed a tool for data amplification based on a 200,000 referee paper data. By using TauJud, it will be able to solve the amplification problem of the judgment document data, and then improve the accuracy of the current judicial prediction model.

## 2 APPROACH

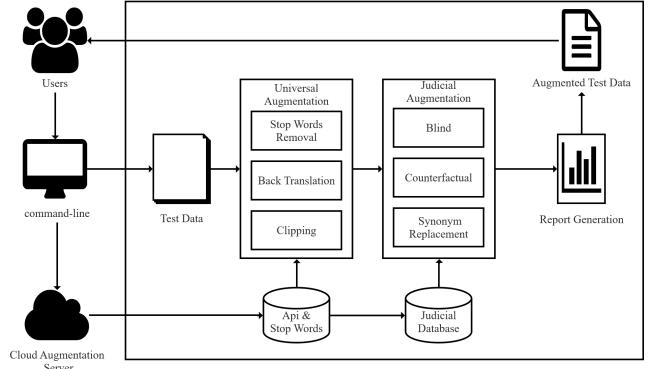
The subject of the test amplification in this article is the referee instrument. As the carrier of the facts of the trial of the case, the judgment documents are described in natural language and expressed in Chinese.

The judicial judgment documents take an important part in the judicial trial and contains the truth about the trial case. Judicial documents have the characteristics of complex format and large amounts of redundancy, which results in much cost of manpower and material resources in the screening of documents. When the judges are dealing with the legal cases, they need to match the referee documents with similar cases in the judicial database for reference.

In judicial prediction, word vectors are often used to parse Chinese text. In this case, there will be more redundant data (usually stop words), which not only causes an unnecessary increase in the volume of the data set, but also leads to the chance of information irrelevant with the trial to participate in decision-making, thus affecting the quality of the model. From the perspective of machine learning, a well-performing model should allow more *important information* to participate in decision-making, while *irrelevant information* does not exist or as little as possible participate in decision-making. Therefore, the removal of stop words and the clipping of text are important operations. In addition, in order to obtain better test augmentation, we hope to be able to transform the vector model. By iterating through the Encoder-Decoder method, the data can complete the transformation without changing its information characteristics. Since this method is often used in machine translation, it is also called back translation.

We divided the *important information* mentioned above into *sensitive attributes* and *non-sensitive attributes*. From a judicial perspective, sensitive attributes describe the basic information of the case but are not relevant to the trial. For example, gender and race are sensitive attributes. In order to make the uneven distribution of sensitive attributes not affect or fairly affect the effect of the dataset, we use the blind method and counterfactual method. Synonym replacement is a common text augmentation method that is classified into a machine learning perspective. However, there are strict requirements for the expression of words in the judicial field, so the scope of synonym mentioned is well-designed.

As can be seen from Fig.1, we provide the framework of TauJud and its three main components. TauJud consists of three main components and add-ons in high-level design. It takes judicial test data as input, and the cloud server is on standby for calling corresponding services. The first component, called universal augmentation, is responsible for the basic methods of text augmentation on machine



**Figure 1: TauJud Workflow**

learning, including stop words removing, back translation and clipping. After this, the second component called judicial augmentation is specifically designed for judicial documents. After two levels of augmentation, TauJud can generate reports of relevant information on test data before and after augmentation. In the following of this section, we describe TauJud's components in detail.

### 2.1 Universal Augmentation

- **Stop Words Removal.** Stop words are words filtered out before processing natural language data. It is a common operation to reduce the disturbance of irrelevant words from the whole document. In TauJud, we adopt *jieba* stop words corpus as a benchmark.
- **Back Translation.** Unlike the back translation method in QANet, we focus on translating sentences into English and back into Chinese, which means it is a one-to-one document augmentation task. We adopt the *Baidu translation API* for translation.
- **Clipping.** Given the complicated structure of judicial documents, we use documents clipping method to ensure the most important parts remaining. We provide an interface for users with custom regular expression to extract appropriate parts from a whole document. In some scenarios, the important paragraph of document can be easily extracted according to a prescribed format and a word library of action words by the local court. It is noteworthy that our dataset(CAIL2018) is composed of causes of action, so there's no need to use clipping method.

### 2.2 Judicial Augmentation

Besides, based on the characteristics of judicature, we develop three possible methods to augment judicial documents. We define a formative judicial document as a five-tuple  $D_m(W_m, c, y, l, p)$  which denotes that judicial document  $D_m$  is mainly composed of the collections of words  $W_m$ , the cause of action  $c$ , the year  $y$ , the location  $l$  and people involved  $p$ . In order to ensure credibility within a community, augmentation must take all these complicated factors into consideration. We construct a judicial corpus with tokens with the following three methods.  $\mathcal{A}$  is designed to modify the original test

**Table 1: Notations**

Symbol	Description
$M$	Number of judicial documents
$m$	Index of a judicial document
$D_m$	$m$ -th judicial document
$c$	Cause of action of a judicial document
$l$	Location of a judicial document
$y$	Year of a judicial document
$p$	People involved of a judicial document
$\mathcal{A}$	Collection of groups of counterfactual tokens as followings
$W_m$	Collection of counterfactual words in judicial document $D_m$

documents, including three groups of counterfactual tokens (year, location, people). For convenience, we define the custom data formats and definitions used in Table 1.

- **Blindness.** Blindness substitutes all tokens with a special identity token, which shows identity term in test data. In TauJud, for document  $D_m$ , blindness can be defined as:

$$D_m(W_m, c, y, l, p) \rightarrow D'_m(W_m, c, b_y, b_l, b_p)$$

where  $b_y, b_l, b_p$  represent fixed symbols as 'YEAR', 'LOCATION' and 'PERSON'.

- **Counterfactual Augmentation.** Counterfactual augmentation does not make the model blind to the recognition items, it uses the generated counterfactual instances to increase the training set of the model. The additional example aims to guide the model to keep the perturbed identity term unchanged. This is a standard technology in computer vision, which is used to keep the model unchanged to the target position, image direction, etc. Counterfactual examples are labeled the same as the original. To guarantee counterfactual token fairness, TauJud can deal with complicated counterfactuals that involve more than just one token substitutions, e.g. "2011, Hua went to Beijing." and "2017 Guo went to Nanjing." The counterfactual augmentation is defined as follows:

$$D_m(W_m, c, y, l, p) \rightarrow D'_m(W_m, c, y', l', p')$$

where  $y' \in \mathcal{A}_Y$ ,  $l' \in \mathcal{A}_L$  and  $p' \in \mathcal{A}_P$ . For  $M$  test judicial documents,  $X$  denotes a group of counterfactual tokens. We can get the general distribution of counterfactual tokens:

$$\forall x_m \in X, x'_m \sim \text{Unif}[\mathcal{A}(x_m)]$$

- **Synonym Replacement.** Randomly choose words from the sentence except stop words. Replace each of these words with one of its synonyms chosen at random. In Judicial field, considering the high sensitivity of legal term, we deploy a large number of pairs of neutral synonyms  $Syn$  and store them as key-value pairs in the database. When  $W_m$  in keys, TauJud queries the database on a cloud server to get the synonym.

### 2.3 Report Generation

We have deployed TauJud as a command-line tool which can be used offline on personal computers as well. In order to help users

have a deeper insight into the process of augmentation and data itself, TauJud implements visual figures based on the augmented documents.

## 3 EVALUATION

In order to verify the usability and applicability of our augmented documents in real-world scenarios, in this section, we design an experiment for evaluation. We perform experiments on the law case dataset CAIL2018, which contains 204,231 test documents in total [9]. All of the test documents are the type of criminal cases, in JSON format. They contain the text of the judicial case and corresponding labels of the term of imprisonment and penalty.

According to the study of Lottier and Stuart [7], the time and space distribution of criminal cases is significant. Our judicial database involves all the cities and provinces of Chinese mainland. For document  $D_m$ , if  $W_m$  is a region in the database, the province of this region can be taken into account for analysis. Besides, if  $W_m$  represents one year, TauJud will randomly generate a new year between 1990 and 2018 as an alternative. We use counterfactual augmentation combined with stop words removal in universal augmentation.

After counterfactual augmentation and back-translation, we get the new province distribution of documents. The province distribution meets uniform distribution after augmentation. 7,500 cases (the number of generated cases decided by the input dataset) are evenly distributed in every province of the mainland. The augmented documents provide a more reasonable regional distribution which can help model take the comprehensive evaluation.

Also, we conduct an analysis on year distribution of test documents, as shown in Fig. 2. CAIL2018 test data contains most of the cases between 2010 and 2018. However, the number of cases from 1990 to 2004 is rather limited. After augmentation, data appears uniform distribution over the year from 1990 to 2018. About 40,000 cases can be found in the dataset in each five-year. In this way, test documents can have more comprehensive coverage of China's judicial year.

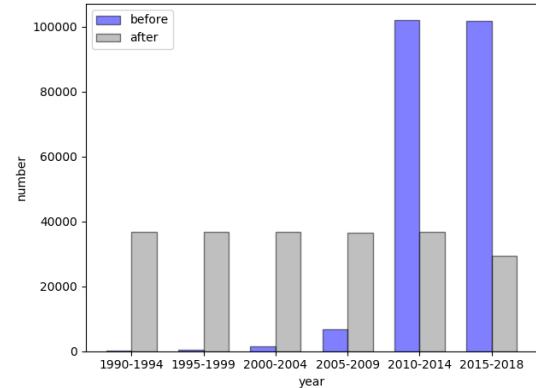
**Figure 2: Year distribution before and after augmentation**

Table 2 takes an example of the augmentation. If sentences are significantly changed, however, the original class labels may no longer be valid. For convenience, we show document  $D_{12428}$  in

**Table 2: Examples of augmentation**

D12428	Before	After
c	In 2017, the defendants Zhao and Zhang quarreled at the west side of the gate of Nanjing Environmental Protection Bureau due to parking problems. Zhao beat Zhang and injured him.	In 2004, the defendants Hu and Guo had words with each other at the west side of the gate of Changsha Environmental Protection Bureau due to parking problems. Hu beat Guo and hurt him.
l	Nanjing	Changsha
y	2017	2004
p	(Zhao, zhang)	(Hu, Guo)

**Table 3: The evaluation of ML models.**

model	n=1	n=5	n=10	n=20
CNN	0.82	0.78	0.61	0.58
Text-CNN	0.52	0.71	0.67	0.65
ATT-CNN	0.83	0.73	0.62	0.59
ResNet	0.50	0.31	0.55	0.54
SVM	0.27	0.66	0.39	0.32
LinearSVM	0.44	0.29	0.29	0.25
RandomForest	0.15	0.57	0.34	0.31

English. After judicial augmentation, the cause of this judicial document is not changed, but it appears in different time, location and people without changing the raw labels, which corresponds to the real world very well. It should be noted that TauJud can also deal with counterfactual tokens such as colour, sex and organization.

To simulate the lack of test data situation, we initiate test data of only 10,000 documents and augment test data for evaluation as a contrast. In order to achieve certain results, we take all of the following approaches: Stop Words Removal, Back Translation, Counterfactual Augmentation and Synonym Replacement. In cf, we simulate real crime distribution instead of random distribution of location of China. We further performed a comparative study on the widely used 7 widely used ML models for real-world penalty prediction analysis on more than two million judicial documents, including both classic machine learning models (SVM, LinearSVM, RandomForest) and deep learning models (CNN, text-CNN, attention-CNN, and ResNet). To evaluate the performance of our ML models, we use accuracy rate as the basic evaluation metric. We augment test documents with TauJud for  $n$  times,  $n = 1, 5, 10, 20$ .

As we can see in Table 3, when the amount of test documents is small ( $n = 1, 5$ ), the result turns out to be very volatile, which means the test data can't show the precise performance of ML models. When  $n$  is large ( $n = 10, 20$ ), it is found that the numerical solutions are convergent and are in good coincidence with experimental results, which proves that TauJud can augment test data for evaluation effectively.

## 4 USER MANUAL

To augment test judicial documents, run "python3 TauJud.py -<arg> -o <arg> -jud <arg> -uni <arg> [-c <arg>]".

- "-h, -help": This argument is optional and is used if users want to get concrete usage of all the arguments.
- "-i, -input": This argument is required. The argument should be a file in JSON format as test data input.
- "-o, -output": This argument is required. Pass the name of the output file in JSON format as the argument.
- "-jud, -judicial": This argument is required. Only three parameters (blind, cf, sysnonyms) can be selected.
- "-uni, -universal": This argument is required. Only two parameters (backtrans, stopworddel) can be selected.
- "-c, -clipping": This argument is optional and is used when users want to select some paragraphs of the document. The scope should satisfy the regular expression.

For each run, for each iteration, the tool provides: (1) total running time; (2) total number of documents; (3) multiple combinations of argumentation; (4) province distribution chart; (5) year distribution.

## 5 CONCLUSION

In this demo paper, we present TauJud, a useful tool to realize the test augmentation of machine learning models in judicial documents. We have shown that judicial data augmentation operations can follow the uniform distribution over time and location, which simulates a large number of judicial cases in the real world. Continued work on TauJud could explore using ELMO embeddings for context preservation. Another thing to explore would be the performance of the newly generated test data to be trained on sequence models(LSTM, BiLSTM, etc.). We hope that TauJud's simplicity makes judicial test data augmentation easy and saves time and cost for labelling data.

## ACKNOWLEDGEMENTS

This work was partially supported by the Fundamental Research Funds for the Central Universities (14380023).

## REFERENCES

- [1] Daniel Berrar and Werner Dubitzky. 2019. Should significance testing be abandoned in machine learning? *IJDSA* 7, 4 (2019), 247–257.
- [2] Reuben Binns. [n.d.]. Fairness in Machine Learning: Lessons from Political Philosophy. ([n. d.]). arXiv:1712.03586 <http://arxiv.org/abs/1712.03586>
- [3] Sahaj Garg, Vincent Perot, Nicole Limtiaco, Ankur Taly, Ed H. Chi, and Alex Beutel. [n.d.]. Counterfactual Fairness in Text Classification through Robustness. ([n. d.]). arXiv:1809.10610 <http://arxiv.org/abs/1809.10610>
- [4] Zichen Guo, Tieke He, Zemin Qin, Zicong Xie, and Jia Liu. 2019. A Content-Based Recommendation Framework for Judicial Cases. In *ICPCSEE*. Springer, 76–88.
- [5] Tie-Ke He, Hao Lian, Ze-Min Qin, Zhen-Yu Chen, and Bin Luo. 2018. PTM: A Topic Model for the Inferring of the Penalty. *JCST* 33, 4 (2018), 756–767.
- [6] Michael Kamp. 2019. *Black-Box Parallelization for Machine Learning*. Ph.D. Dissertation. Universitäts-und Landesbibliothek Bonn.
- [7] Stuart Lottier. [n.d.]. Distribution of Criminal Offenses in Metropolitan Regions. 29, 1 ([n. d.]), 37. <https://doi.org/10.2307/1137347>
- [8] Jason Wei and Kai Zou. [n.d.]. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. ([n. d.]). arXiv:1901.11196 <http://arxiv.org/abs/1901.11196>
- [9] Chaojun Xiao, Haoxi Zhong, Zhipeng Guo, et al. 2018. Cail2018: A large-scale legal dataset for judgment prediction. *arXiv preprint arXiv:1807.02478* (2018).
- [10] Zihuan Xu, Tieke He, Hao Lian, Jiabing Wan, and Hui Wang. 2019. Case Facts Analysis Method Based on Deep Learning. In *WISA*. Springer, 92–97.
- [11] Ge Yan, Yu Li, Shu Zhang, and Zhenyu Chen. 2019. Data Augmentation for Deep Learning of Judgment Documents. In *ISCIIDE*. Springer, 232–242.
- [12] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. [n.d.]. QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension. ([n. d.]). arXiv:1804.09541 <http://arxiv.org/abs/1804.09541>

# EShield: Protect Smart Contracts against Reverse Engineering

Wentian Yan

yanwentian@pku.edu.cn

HCST, CS, Peking University

Beijing, China

Jianbo Gao

Zhenhao Wu

gaojianbo@pku.edu.cn

zhenhao@pku.edu.cn

HCST, CS, Peking University

Beijing, China

Boya Blockchain Inc.

Beijing, China

Yue Li

liyue\_cs@pku.edu.cn

HCST, CS, Peking University

Beijing, China

Zhi Guan\*

guan@pku.edu.cn

National Engineering Research  
Center for Software Engineering,

Peking University

Beijing, China

Qingshan Li

liqs@pku.edu.cn

HCST, CS, Peking University

Beijing, China

Boya Blockchain Inc.

Beijing, China

Zhong Chen

zhongchen@pku.edu.cn

HCST, CS, Peking University

Beijing, China

## ABSTRACT

Smart contracts are the back-end programs of blockchain-based applications and the execution results are deterministic and publicly visible. Developers are unwilling to release source code of some smart contracts to generate randomness or for security reasons, however, attackers still can use reverse engineering tools to decompile and analyze the code. In this paper, we propose EShield, an automated security enhancement tool for protecting smart contracts against reverse engineering. EShield replaces original instructions of operating jump addresses with anti-patterns to interfere with control flow recovery from bytecode. We have implemented four methods in EShield and conducted an experiment on over 20k smart contracts. The evaluation results show that all the protected smart contracts are resistant to three different reverse engineering tools with little extra gas cost.

## CCS CONCEPTS

- Security and privacy → Software reverse engineering; Software security engineering;
- Theory of computation → Program analysis.

## KEYWORDS

Reverse Engineering, Smart Contract, Blockchain, Ethereum, Program Analysis

\*Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404365>

## ACM Reference Format:

Wentian Yan, Jianbo Gao, Zhenhao Wu, Yue Li, Zhi Guan, Qingshan Li, and Zhong Chen. 2020. EShield: Protect Smart Contracts against Reverse Engineering. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404365>

## 1 INTRODUCTION

Smart contracts are programs running on blockchain and can be invoked through transactions. Due to the irreversible nature of transactions, blockchain-based applications have been widely adopted in recent years, covering various scenarios including games, gambling, finance, etc [5]. All the smart contracts deployed on blockchain are publicly visible and the execution of transactions are deterministic, which makes it difficult to generate random numbers in Ethereum for game and gambling applications [1, 11]. Therefore some applications have to generate pseudo-random numbers through complex operations and make the smart contracts with key random functions closed source. However, the contracts can be analyzed using reverse engineering tools (e.g. Erays [13]) and players can utilize the information to increase the chance of breeding profitable kitties, which may destroy randomness of the game and lead to a reduction in the value of rare kitties. Except for generating randomness, developers also make the contracts closed source for security reasons to prevent attackers from analyzing, but the contracts still can be decompiled and exploited by analyzing tools [7, 10].

Although anti-reverse engineering has become a strong need of developers, there are few effective tools for smart contracts. Due to the special architecture of Ethereum, the existing anti-reverse engineering techniques on other platforms such as anti-debugging and obfuscation [4, 8] are either ineffective or too expensive for Ethereum smart contracts. The main challenges of protecting smart contracts against reverse engineering are as follows.

**Challenge 1. Non Von Neumann Virtual Machine.** The Ethereum virtual machine does not follow the standard Von Neumann architecture and the code of smart contracts is stored in

a virtual ROM rather than generally-accessible memory or storage [11]. Therefore the code cannot be modified during execution, which makes it impossible to execute encrypted code in Ethereum.

**Challenge 2. Gas Cost.** Gas is the fuel of computation in Ethereum and users are charged according to the executed bytecode. Obfuscation requires adding a large amount of extra instructions into bytecode and leads to a significant increase in gas cost, which makes it unacceptable to developers and users.

**EShield Solution.** To address the aforementioned challenges, we have developed EShield for protecting smart contracts against reverse engineering. The key insight behind EShield is to increase the difficulty of recovering control flow graph (CFG) by interfering with identifying the connections between basic blocks.

EShield first builds the CFG for smart contracts, then modifies the bytecode and inserts four anti-patterns into basic blocks to ensure the modified bytecode is equivalent to the original bytecode, finally generates the protected bytecode of smart contracts. We conducted experiments on over 20K smart contracts and all the protected bytecode cannot be decompiled by existing reverse engineering tools.

## 2 OVERVIEW

The general work flow of EShield is shown in Figure 1. In detail that EShield consists of four components. It takes runtime bytecodes of Ethereum smart contract as input and put them into an analyzer. The analyzer recovers the CFG and detects the suitable position to insert the anti-patterns. The information will be transmitted to the original constructor where the original bytecode will be reconstructed to be suitable to insert the anti-patterns. At the same time, the pattern constructor produces the anti-patterns with the obfuscated jump address. In the end, the bytecode constructor will insert the anti-patterns into the reconstructed original bytecodes and generates the pattern bytecodes.

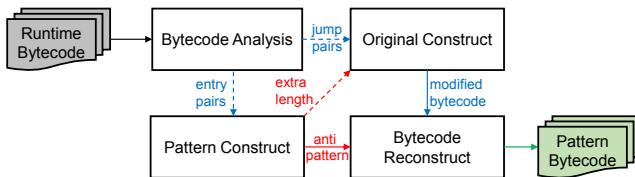


Figure 1: The framework of EShield

### 2.1 Bytecode Analysis

EShield first uses the evm cfg-builder to recover the CFG of smart contract bytecodes. It marks every JUMP or JUMPI instruction with their jump addresses and keeps them in a key-values pairs which named as jump-pairs. The analyzer will check all the jump-pairs and add other three values in each pair. First value is the function address of JUMP(I), second is the position between JUMP(I) and its jump address(the address of JUMP(I) is ahead of or behind the jump address). Third is the length of jump address. These three values are important in the step of original construct.

The control flow of smart contract can be split into blocks and each block represents a function. The entry can be recognized

by an sequence of instructions such as EQ-PUSH-JUMPI. The entry detector will analyze all the instructions of entry and mark JUMPI with its function address, as we called entry-pairs. Every entry can be used to insert the anti-patterns to generate the security enhanced bytecode.

### 2.2 Anti-pattern Construct

**Pattern 1.** The operations in memory cannot be handled well in current reverse tools, so that it can be utilized to obfuscate the jump address. EShield use MSTORE to write the jump address into memory and then use MLOAD to load it into stack. The instruction JUMP[I] will load the function address and jump to the entry of the target function as normal. The instructions can be executed accurately but the reverse tools cannot analyze the real jump address.

**Pattern 2.** Pattern 2 is similar with pattern 1. It utilizes the storage to handle the jump address. The SSTORE instruction stores the function address into storage and SLOAD loads it into stack, which can be used to execute JUMP[I] accurately.

**Pattern 3.** EShield uses SHA3 and a series of operations to get the function address. SHA3 is also a troublesome instruction which the current reverse tools can not handle. They usually generate a formula to represent the execution result of SHA3 and won't calculate the value. EShield uses a clever but simple method to generate function address by SHA3. EShield uses SHA3 to generate two identical hash strings and divide the two value to get 1. Then multiplies the address and 1 to get the final address. Inspired by this idea we can use more complex methods to generate the function address.

**Pattern 4.** Using the call value of external smart contract to produce the jump address is also an effective method to fight against the reverse analysis. We prepared a specific external contract which can return a fixed value by a CALL. Then EShield utilizes the fixed value to produce the jump address just like the way in pattern 3.

All the patterns above will increase the length of the bytecode, so the pattern constructor will transmit the number of extra length to original constructor.

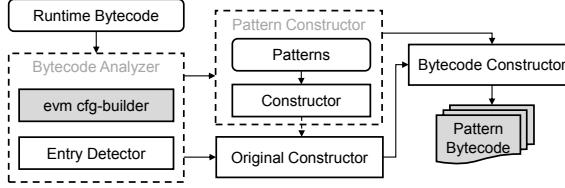
### 2.3 Original Construct

With the extra length and jump-pairs, original constructor will reconstruct the original bytecode to make it suitable for the anti-patterns. There are three steps to finish the entire process of reconstruction. Step 1, if the jump address is behind the position of entry, the constructor will add extra length and the address value. Step 2, calculate all the extra length produced by increasing length of jump address. Maybe the length of original jump address is only 1 byte, then after adding the extra length of pattern it comes to 2 bytes. This situation will probably lead to an increase in its own length of address. The last step, calculate the change of length in other addresses which result from the situation in step 2.

## 3 USING ESHIELD

As illustrated in Figure 2, EShield can be divided into four parts in implementation, bytecode analyzer, pattern constructor, original constructor and bytecode constructor. Batches of Ethereum runtime bytecodes can be saved in files and passed into EShield as input.

User can select which pattern will be used to protect smart contract. In the end, EShield will generate a single pattern bytecode file for each pattern the user selected.



**Figure 2: The architecture of EShield in implementation**

**Bytecode Analyzer.** Bytecode analyzer consists of two components, evm-cfg-builder<sup>1</sup> and entry detector. The evm-cfg-builder is a reliable foundation for building program analysis tools in EVM. It extracts the CFG of bytecode so that EShield can mark all the JUMP(I) with jump addresses and other necessary values to generate the jump-pairs, then transmits them to original constructor. Entry detector will recognize and mark every entry of function, then transmit them to the pattern constructor.

**Pattern Constructor.** Pattern constructor receives the entry pairs from bytecode analyzer and generates the anti-patterns for each pattern. The jump address in every anti-pattern will be modified to be suitable for inserting. The anti-patterns will be transmitted to the bytecode constructor and the extra length of anti-pattern will be passed into original constructor.

**Original Constructor.** After receiving the original bytecode and extra length, original constructor begins to reconstruct the bytecode. It will consider all the situations that can make the jump address change and modify them correctly. The modified bytecode will be transmitted to the final constructor to produce the protected bytecodes.

**Bytecode Constructor.** Bytecode constructor is the final constructor to generate the pattern bytecodes. It will insert the anti-patterns into the modified bytecode to replace the old entry bytecode. EShield will generate a single file for each chosen pattern.

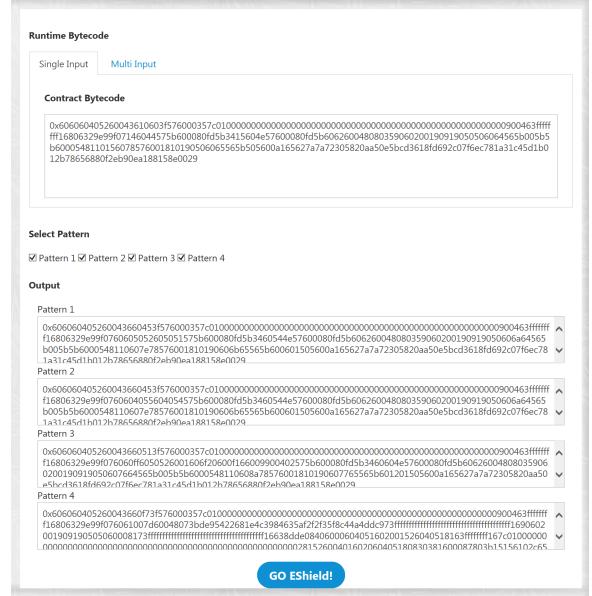
EShield has been deployed as a web service and the command-line version with the same functions is also available. Figure 3 shows the web page of EShield. Users can input the bytecodes or the bytecode file path. After selecting the patterns and setting the output path, users can generate the pattern bytecodes or files by clicking the GO button. At last the pattern bytecodes will be displayed in the page, or saved into files.

## 4 PRELIMINARY EVALUATION

We downloaded the runtime bytecode of 20,266 smart contracts deployed on Ethereum mainnet and used EShield to generate four different patterns of protected bytecodes for each smart contract. The experiments were conducted on a virtual machine with 1 vCPU, 6G RAM and Ubuntu 18.04 as the OS.

To evaluate the effectiveness of EShield, we chose three reverse tools, **Erays** [13], **Vandal** [2, 7] and **Gigahorse** [6] as benchmarks.

<sup>1</sup>[https://github.com/crytic/evm\\_cfg\\_builder/](https://github.com/crytic/evm_cfg_builder/)



**Figure 3: EShield Screenshot**

These reverse tools take runtime bytecode as input and recover CFG and generate other forms of high-level representations, which are the state of art reverse analysis tools of smart contracts.

In addition, we use the following criteria to evaluate the result of reversing analysis, *i.e.* Partially Failed (PF) and Entirely Failed (EF). If a reverse tool generates a wrong result, we consider this as a PF case and the rate of PF cases in the dataset is PFR; if the reverse tool cannot generate any results, it is called a EF case and the rate is EFR.

The way to determine the correctness depends on the results generated by the three reverse tools. **Erays** converts runtime bytecode to a CFG in PDF format. If Erays cannot analyze the jump address, it will miss a function block and create fewer PDF files than original one, this situation is a PF case. If there is no PDF file, it is an EF case. **Vandal** converts runtime bytecode into semantic logic relations and CFG. If Vandal cannot find the jump address, it will miss a block and report an error message, which is called a PF case. If there is no result, it is called an EF case. **Gigahorse** is the same as Vandal.

As shown in Table 1, all the four patterns can effectively fight against the reverse tools. The total failed rate (PFR + EFR) of each pattern achieves 100%, no matter which reverse tool was selected. The results show the effectiveness of EShield.

Since different reverse tools have different forms of representations to record the control flow, they have different ways to deal with the failure of CFG recovery. Some reverse tools will terminate directly when there is an error, but others may not. That's why the PFRs and EFRs are so different between different reverse tools.

For each reverse tool, the experiment results are similar even though the patterns are different. It strongly proved that all the four anti-patterns are effective and EShield is enable to fight against reverse analysis.

**Table 1: Evaluation Results. P: Patterns**

P	Tools	PF cases (PFR)	EF cases (EFR)	Total
1	Erays	272 (1.34%)	19994 (98.66%)	<b>20266 (100%)</b>
	Vandal	19508 (96.26%)	758 (3.74%)	<b>20266 (100%)</b>
	Gigahorse	20266 (100%)	0 (0%)	<b>20266 (100%)</b>
	<b>Average</b>	<b>13349 (65.87%)</b>	<b>6917 (34.13%)</b>	<b>20266 (100%)</b>
2	Erays	272 (1.34%)	19994 (98.66%)	<b>20266 (100%)</b>
	Vandal	19515 (96.29%)	751 (3.71%)	<b>20266 (100%)</b>
	Gigahorse	20266 (100%)	0 (0%)	<b>20266 (100%)</b>
	<b>Average</b>	<b>13351 (65.88%)</b>	<b>6915 (34.12%)</b>	<b>20266 (100%)</b>
3	Erays	272 (1.34%)	19994 (98.66%)	<b>20266 (100%)</b>
	Vandal	19512 (96.28%)	754 (3.72%)	<b>20266 (100%)</b>
	Gigahorse	20266 (100%)	0 (0%)	<b>20266 (100%)</b>
	<b>Average</b>	<b>13350 (65.87%)</b>	<b>6916 (34.13%)</b>	<b>20266 (100%)</b>
4	Erays	272 (1.34%)	19994 (98.66%)	<b>20266 (100%)</b>
	Vandal	19517 (96.30%)	749 (3.70%)	<b>20266 (100%)</b>
	Gigahorse	20266 (100%)	0 (0%)	<b>20266 (100%)</b>
	<b>Average</b>	<b>13351 (65.88%)</b>	<b>6915 (34.12%)</b>	<b>20266 (100%)</b>

**Table 2: Cost of Using EShield. P: Patterns, Extra<sub>Create</sub>: Extra gas cost of Creating protected contracts, Extra<sub>Dollars</sub>: Extra cost in dollars, Extra<sub>Call</sub>: Extra gas cost of Calling protected contracts, Extra<sub>Dollars</sub>: Extra cost in dollars, Time: average Time cost of running EShield.**

P	Extra <sub>Create</sub>	Extra <sub>Dollars</sub>	Extra <sub>Call</sub>	Extra <sub>Dollars</sub>	Time
1	1200	0.0084	21	0.0001	1.20s
2	1200	0.0084	20212	0.1418	1.19s
3	3600	0.0253	82	0.0006	1.22s
4	37636	0.2640	33297	0.2336	1.32s

Table 2 illustrated the average extra consumption of gas in each pattern. We can exchange the gas to dollar by the time of 9th April in 2020. As we use the default gas price(1 gas unit=41 Gwei, which is much higher than usual) to calculate that 1000 gas worth \$0.00701551. That means when user creates or calls a protected smart contract, in the worst case it will cost only \$0.2640 and \$0.2336 respectively. In the best case that creating a smart contract protected by pattern 1 and 2 will only cost about \$0.0084, and calling a contract protected by pattern 1 and 3 will only cost about \$0.0001 and \$0.0006 respectively. The extra cost of using EShield is very low and acceptable to both developers and users. Table 2 also shows the average time EShield spends in generating protected contracts. The average time it costs was between 1.19s and 1.32s. Considering the average size of original bytecode we used is 6.46KB, these results show that EShield is efficient enough to be used in practice.

## 5 RELATED WORK

Research on reverse engineering and anti-reversing has been in progress for decades and many approaches focusing on different platforms were proposed. As mentioned above, there are some papers about decompiling and analyzing EVM bytecode as well, however, to the best of our knowledge, there is no previous work on protecting smart contracts against reverse engineering.

Porosity [9] is the first reverse engineering tool for Ethereum smart contracts, which can generate readable Solidity-like source code from EVM bytecode. It is removed from benchmarks in our

experiments due to the high failure rate. Madmax [7] is a static program analysis tool for detecting gas-focused vulnerabilities in smart contracts and it uses Vandal [2] to decompile bytecode. Gigahorse [6] is an alternative to Vandal and performs better in decompiling unmodified bytecode.

There are some anti-reversing techniques available on other platforms. Software on Windows platform often use anti-virtualization and anti-debugging to prevent analyzing [3]. Android developers also use packers to encrypt bytecode and decrypt it to memory at runtime for protection [12]. However, these approaches cannot be applied to the smart contract due to the different architectures.

## 6 CONCLUSION

In this demo paper, we presented EShield, a security enhancement tool for protecting Ethereum smart contracts against reverse engineering. EShield performs bytecode replacement with four different patterns to interfere with control flow graph recovery. EShield managed to protect all the smart contracts from three reverse engineering tools with little extra cost in the evaluation. In the future, we plan to explore more methods and generalize the technique to other blockchain platforms.

## ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under the grant No. 61672060.

## REFERENCES

- [1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*. Springer, 164–186.
- [2] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).
- [3] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 177–186.
- [4] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. 2007. Software protection through anti-debugging. *IEEE Security & Privacy* 5, 3 (2007), 82–84.
- [5] Jianbo Gao, Han Liu, Yue Li, Chao Liu, Zhiqiang Yang, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. Towards automated testing of blockchain-based decentralized applications. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 294–299.
- [6] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1176–1186.
- [7] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116.
- [8] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *NDSS*.
- [9] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25 (2017), 11.
- [10] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Security: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.
- [11] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
- [12] Lei Xue, Xiapi Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 358–369.
- [13] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum’s opaque smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1371–1385.

# Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts

Gustavo Grieco  
Trail of Bits, USA

Will Song  
Trail of Bits, USA

Artur Cygan  
Trail of Bits, USA

Josselin Feist  
Trail of Bits, USA

Alex Groce  
Northern Arizona University, USA

## ABSTRACT

Ethereum smart contracts—autonomous programs that run on a blockchain—often control transactions of financial and intellectual property. Because of the critical role they play, smart contracts need complete, comprehensive, and effective test generation. This paper introduces an open-source smart contract fuzzer called Echidna that makes it easy to automatically generate tests to detect violations in assertions and custom properties. Echidna is easy to install and does not require a complex configuration or deployment of contracts to a local blockchain. It offers responsive feedback, captures many property violations, and its default settings are calibrated based on experimental data. To date, Echidna has been used in more than 10 large paid security audits, and feedback from those audits has driven the features and user experience of Echidna, both in terms of practical usability (e.g., smart contract frameworks like Truffle and Embark) and test generation strategies. Echidna aims to be good at finding real bugs in smart contracts, with minimal user effort and maximal speed.

## CCS CONCEPTS

- Software and its engineering → Dynamic analysis; Software testing and debugging.

## KEYWORDS

smart contracts, fuzzing, test generation

### ACM Reference Format:

Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404366>

## 1 INTRODUCTION

Smart contracts for the Ethereum blockchain [5], usually written in the Solidity language [25], facilitate and verify high-value financial transactions, as well as track physical goods and intellectual property. Thus, it is essential that these programs be correct and secure, which is *not* always the case [4]. Recent work surveying and categorizing flaws in critical contracts [11] established that

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

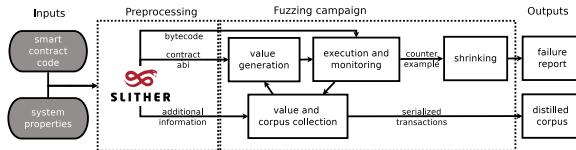
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404366>

fuzzing using custom user-defined properties might detect up to 63% of the most severe and exploitable flaws in contracts. This suggests an important need for high-quality, easy-to-use fuzzing for smart contract developers and security auditors. Echidna [23] is an open-source Ethereum smart contract fuzzer. Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities during fuzzing campaigns, Echidna supports three types of properties: (1) user-defined properties (for property-based testing [7]), (2) assertion checking, and (3) gas use estimation. Currently, Echidna can test both Solidity and Vyper smart contracts, and supports most contract development frameworks, including Truffle and Embark. Echidna has been used by Trail of Bits for numerous code audits [11, 24]. The use of Echidna in internal audits is a key driver in three primary design goals for Echidna. Echidna must (1) be easy to use and configure; (2) produce good coverage of the contract or blockchain state; and (3) be *fast* and produce results quickly.

The third design goal is essential for supporting the first two design goals. Tools that are easy to run and produce quick results are more likely to be integrated by engineers during the development process. This is why most property-based testing tools have a small default run-time or number of tests. Speed also makes use of a tool in continuous integration (CI) (e.g., <https://cryptic.io>) more practical. Finally, a fast fuzzer is more amenable to experimental techniques like mutation testing [21] or using a large set of benchmark contracts. The size of the statistical basis for decision-making and parameter choices explored is directly limited by the speed of the tool. Of course, Echidna supports lengthy testing campaigns as well: there is no upper bound on how long Echidna can run, and with coverage-based feedback there is a long-term improvement in test generation quality. Nonetheless, the goal of Echidna is to reveal issues to the user in less than 5 minutes.

Fuzzing smart contracts introduces some challenges that are unusual for fuzzer development. First, a large amount of engineering effort is required to represent the semantics of blockchain execution; this is a different challenge than executing instrumented native binaries. Second, since Ethereum smart contracts compute using transactions rather than arbitrary byte buffers, the core problem is one of transaction sequence generation, more akin to the problem of unit test generation [20] than traditional fuzzing. This makes it important to choose parameters such as the length of sequences [3] that are not normally as important in fuzzing or in single-value-generation as in Haskell's QuickCheck [7]. Finally, finding smart contract inputs that cause pathological execution times is not an exotic, unusual concern, as in traditional fuzzing [17]. Execution on the Ethereum blockchain requires use of gas, which has a price in cryptocurrency. Inefficiency can be costly, and malicious inputs can lock a contract by making all transactions require more gas than a transaction is allowed to use. Therefore producing quantitative



**Figure 1: The Echidna architecture.**

output of maximum gas usage is an important fuzzer feature, alongside more traditional correctness checks. Echidna incorporates a worst-case gas estimator into a general-purpose fuzzer, rather than forcing users to add a special-purpose gas-estimation tool [2, 18] to their workflow.

## 2 ARCHITECTURE AND DESIGN

### 2.1 Echidna Architecture

Figure 1 shows the Echidna architecture divided into two steps: pre-processing and the fuzzing campaign. Our tool starts with a set of provided contracts, plus properties integrated into one of the contracts. As a first step, Echidna leverages Slither [10], our smart contract static analysis framework, to compile the contracts and analyze them to identify useful constants and functions that handle Ether (ETH) directly. In the second step, the fuzzing campaign starts. This iterative process generates random transactions using the application binary interface (ABI) provided by the contract, important constants defined in the contract, and any previously collected sets of transactions from the corpus. When a property violation is detected, a counterexample is automatically minimized to report the smallest and simplest sequence of transactions that triggers the failure. Optionally, Echidna can output a set of transactions that maximize coverage over all the contracts.

### 2.2 Continuous Improvement

A key element of Echidna’s design is to make continuous improvement of functionality sustainable. Echidna has an extensive test suite (that checks detection of seeded faults, not just proper execution) to ensure that existing features are not degraded by improvements, and uses Haskell language features to maximize abstraction and applicability of code to new testing approaches.

*Tuning Parameters.* Echidna provides a large number of configurable parameters that control various aspects of testing. There are currently more than 30 settings, controlled by providing Echidna with a YAML configuration file. However, to avoid overwhelming users with complexity and to make the out-of-the-box experience as smooth as possible, these settings are assigned carefully chosen default values. Default settings with a significant impact on test generation are occasionally re-checked via mutation testing [12] or benchmark examples to maintain acceptable performance. This, like other maintenance, is required because other functionality changes may impact defaults. For example, changes in which functions are called (e.g., removing view/pure functions with no assertions) may necessitate using a different default sequence length. Parameter tuning can produce some surprises with major impact on users: e.g., the dictionary of mined constants was initially only used infrequently in transaction generation, but we found that mean coverage

on benchmarks could be improved significantly by using constants a full 40% of the time.

## 3 USAGE

Before starting Echidna, the smart contract to test should have either explicit echidna\_ properties (public methods that return a Boolean have no arguments) or use Solidity’s assert to express properties. For instance, Figure 2a shows a contract with a property that tests a simple invariant. After defining the properties to test, running Echidna is often as simple as installing it or using the provided Docker image and then typing:

```
$ echidna-test Contract.sol --contract TEST
```

An optional YAML configuration file overriding default settings can be provided using the –config option. Additionally, if a path to a directory is used instead of a file, Echidna will auto-detect the framework used (e.g. Truffle) and start the fuzzing campaign.

By default, Echidna uses a dashboard output similar to AFL’s as shown in Figure 2b. However, a command line option or a config file can change this to output plaintext or JSON. The config file also controls various properties of test generation, such as the maximum length of generated transaction sequences, the frequency with which mined constants are used, whether coverage driven feedback is applied, whether maximum gas usage is computed, and any functions to blacklist from the fuzzing campaign.

## 4 EXPERIMENTAL EVALUATION

### 4.1 Setup

We compared Echidna’s performance to the MythX platform [9], accessed via the solfuzz [19] interface, on a set of reachability targets. Our experiments are produced by insertion of assert(false) statements, on a set of benchmark contracts [1] produced for the VeriSmart safety-checker [22]. To our knowledge, MythX is the only comparable fuzzer that supports arbitrary reachability targets (via supporting assertion-checking). Comparing with a fuzzer that only supports a custom set of built-in detectors, such as ContractFuzzer [16], which does not support arbitrary assertions in Solidity code, is difficult to do objectively, as any differences are likely to be due to specification semantics, not exploration capability. MythX is a commercial SaaS platform for analyzing smart contracts. It offers a free tier of access (limited to 5 runs/month, however) and can easily run assertion checking on contracts via solfuzz, which provides an interface similar to Echidna’s. MythX analyzes the submitted contracts using the Mythril symbolic execution tool [8] and the Harvey fuzzer [26]. Harvey is a state-of-the-art closed-source tool, with a research paper describing its design and implementation in ICSE 2020 [26]. We also attempted to compare to the ChainFuzz [6] tool; unfortunately, it is not maintained, and failed to analyze contracts, producing an error reported in a GitHub issue submitted in April of 2019 (<https://github.com/ChainSecurity/ChainFuzz/issues/2>).

### 4.2 Datasets

*VeriSmart.* To compare MythX and Echidna, we first analyzed the contracts in the VeriSmart benchmark [1] and identified all contracts such that 1) both tools ran on the contract and 2) neither tool

```

contract TEST {
    bool flag0; bool flag1;

    function set0(int val) public returns (bool) {
        if (val % 100 == 23) { flag0 = true; }
    }

    function set1(int val) public returns (bool) {
        if (val % 10 == 5 && flag0) { flag1 = true; }
    }

    function echidna_flag() public returns (bool) {
        return(!flag1);
    }
}

```

(a) A contract with an echidna property.

Echidna 1.4.0.1

Tests found: 1  
Unique instructions: 242  
Unique codehashes: 1

echidna\_flag: FAILED!

Call sequence:  
1.set0(23)  
2.set1(5)

Campaign complete, C-c or esc to exit

(b) A screenshot of the UI with the result of a fuzzing campaign

Figure 2: Using Echidna to test a smart contract

reported any issues with the contract. This left us with 12 clean contracts to compare the tools’ ability to explore behavior. We inserted `assert(false)` statements into each of these contracts, after every statement, resulting in 459 contracts representing reachability targets. We discarded 44 of these, as the assert was unconditionally executed in the contract’s constructor, so no behavior exploration was required to reach it.

*Tether.* For a larger, more realistic example, we modified the actual blockchain code for the TetherToken contract<sup>1</sup>, and again inserted `assert(false)` targets to investigate reachability of the code. Tether is one of the most famous “stablecoins”, a cryptocurrency pegged to a real-world currency, in this case the US dollar, and has a market cap of approximately 6 billion dollars. The contract has been involved in more than 23 million blockchain transactions.

### 4.3 Results

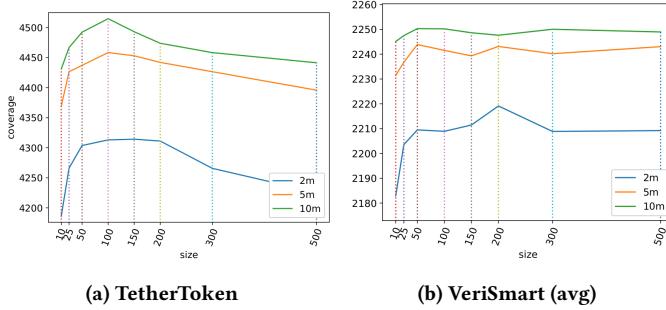
We then ran solfuzz’s default quick check and Echidna with a 2-minute timeout on 40 randomly selected targets. Echidna was able to produce a transaction sequence reaching the assert for 19 of the 40 targets, and solfuzz/MythX generated a reaching sequence for 15 of the 40, all of which were also reached by Echidna. While the time to reach the assertion was usually close to 2 minutes with solfuzz, Echidna needed a maximum of only 52 seconds to hit the hardest target; the mean time required was 13.9 seconds, and the median time was only 6.9 seconds. We manually examined the targets not detected by either tool, and believe them all to represent unreachable targets, usually due to being inserted after an unavoidable return statement, or being inserted in the SafeMath contract, which redefines assert. Of the reachable targets, Echidna was able to produce sequences for 100%, and solfuzz/MythX for 78.9%. For Echidna, we repeated each experiment 10 more times, and Echidna always reached each target. Due to the limit on MythX runs, even under a developer license (500 executions/month), we were unable to statistically determine the stability of its results to the same degree, but can confirm that for two of the targets, a second run succeeded, and for two of the targets three additional runs still failed to reach the assert. Running solfuzz with the `-mode standard` argument (not available to free accounts) did detect all four, but it required at least 15 minutes of analysis time in each

<sup>1</sup><https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code>

case. Figure 4 shows the key part of the code for one of the four targets Echidna, but not solfuzz/MythX (even with additional runs), was able to reach. The assert can only be executed when a call has been made to the approve function, allowing the sender of the transferFrom call to send an amount greater than or equal to `_amount`, and when the contract from which transfer is to be made has a token balance greater than `_amount`. Generating a sequence with the proper set of functions called and the proper relationships between variables is a difficult problem, but Echidna’s heuristic use of small numeric values in arguments and heuristic repetition of addresses in arguments and as message senders is able to navigate the set of constraints easily. A similar set of constraints over allowances and/or senders and function arguments is involved in two of the other four targets where Echidna performs better.

When using the Tether contract, we again randomly selected 40 targets, and ran two minutes of testing on each with solfuzz and Echidna. Echidna was able to reach 28 of the 40 targets, with mean and median runtimes of 24 and 15 seconds, respectively. The longest run required 103 seconds. On the other hand, solfuzz/MythX was unable to reach any of the targets using the default search. MythX-/solfuzz was able to all detect the targets Echidna detected using the standard search, and detected one additional target. The mean time required for detection, however, was almost 16 minutes. The additional target reached by solfuzz involves adding an address to a blacklist, then destroying the funds of that address. Because an address can also be removed from a blacklist, there is no simple coverage-feedback to encourage avoiding this, and there are many functions to test, Echidna has trouble generating such a sequence. However, using a prototype of a swarm testing [13] mode not yet included in the public release of Echidna, but briefly discussed in the conclusion below, we were able to produce such a sequence in less than five minutes. Even without swarm testing, we were able to detect the problem in between 10 and 12 minutes, using a branch (to be merged in the near future) that incorporates more information from Slither, and uses some novel mutation strategies. Of the 11 targets hit by neither tool, we manually determined that all but two are clearly actually unreachable.

As a separate set of experiments, we measured the average coverage obtained on the VeriSmart and Tether token contracts, with various settings for the length of transaction sequences, ranging from very short (length 10) to very long (length 500) for runs of 2,



**Figure 3: Coverage obtained given short runs (2, 5 and 10 minutes) with different transaction sequence lengths.**

```
if (balances[_from] >= _amount
    && allowed[_from][msg.sender] >= _amount
    && _amount > 0
    && balances[_to] + _amount > balances[_to]) {
    balances[_from] -= _amount;
    allowed[_from][msg.sender] -= _amount;
    assert(false);
```

**Figure 4: Code for a difficult reachability target.**

5, and 10 minutes each. Figures 3a and 3b show that the current default value used by Echidna (100) is a reasonable compromise to maximize coverage in short fuzzing campaigns. Each run was repeated 10 times to reduce the variability of such short campaigns.

## 5 RELATED WORK

Echidna inherits concepts from property-based fuzzing, first popularized by the QuickCheck tool [7] and from coverage-driven fuzzing, perhaps best known via the American Fuzzy Lop tool [27]. Other fuzzers for Ethereum smart contracts include Harvey [26], ContractFuzzer [16], and ChainFuzz [6]. We were unable to get ContractFuzzer to produce useful output within a four hour timeout, and ChainFuzz no longer appears to work. Harvey is closed-source, but is usable via the MythX [9] CI platform and the solfuzz [19] tool. Echidna uses information from the Slither static analysis tool [10] to improve the creation of Ethereum transactions.

## 6 CONCLUSIONS AND FUTURE WORK

Echidna is an effective, easy-to-use, and *fast* fuzzer for Ethereum blockchain smart contracts. Echidna provides a potent out-of-the-box fuzzing experience with little setup or preparation, but allows for considerable customization. Echidna supports assertion checking, custom property-checking, and estimation of maximum gas usage—a core feature set based on experience with security audits of contracts. The default test generation parameters of Echidna have been calibrated using real-world experience in commercial audits and via benchmark experiments and mutation analysis. In our experiments, Echidna outperformed a comparable fuzzer using sophisticated techniques: Echidna detected, in less than 2 minutes, many reachability targets that required 15 or more minutes with solfuzz, on both benchmark contracts and the real-world Tether token. Echidna is under heavy active development. Recently added

or in-progress features include gas estimation, test corpus collection, integration of Slither static analysis information, and improved mutation for feedback-driven fuzzing. One future work will add a driver mode, similar to the swarm tool [14] for the SPIN model checker [15], to make better use of configuration diversity, including swarm testing [13], in order to fully exploit multicore machines. In particular, this mode will enable Echidna to produce even more accurate maximum gas usage estimates.

## REFERENCES

- [1] VeriSmart benchmark. <https://github.com/kupl/VeriSmart-benchmarks>.
- [2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts, 2019.
- [3] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Automated Software Engineering*, pages 19–28, 2008.
- [4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts SoK. In *International Conference on Principles of Security and Trust*, pages 164–186, 2017.
- [5] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. <https://github.com/ethereum/wiki/wiki/White-Paper>, 2013.
- [6] Chain Security. <https://github.com/ChainSecurity/ChainFuzz>.
- [7] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.
- [8] ConsenSys. Mythril: a security analysis tool for ethereum smart contracts. <https://github.com/ConsenSys/mythril-classic>, 2017.
- [9] ConsenSys Diligence. <https://mythx.io/>.
- [10] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019.
- [11] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *International Conference on Financial Cryptography and Data Security*, 2020.
- [12] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18*, pages 25–28, New York, NY, USA, 2018. ACM.
- [13] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
- [14] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.
- [15] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
- [16] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.
- [17] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.
- [18] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability, 2019.
- [19] Bernhard Mueller. <https://github.com/b-mueller/solfuzz>.
- [20] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.
- [21] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.
- [22] Sunbeam So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VeriSmart: A highly precise safety verifier for ethereum smart contracts. In *IEEE Symposium on Security & Privacy*, 2020.
- [23] Trail of Bits. Echidna: Ethereum fuzz testing framework. <https://github.com/cryptic/echidna>, 2018.
- [24] Trail of Bits. Trail of bits security reviews. <https://github.com/trailofbits/publications#security-reviews>, 2019.
- [25] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. <http://gavwood.com/paper.pdf>, 2014.
- [26] Valentin Wüstholtz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering*, 2020.
- [27] Michał Zalewski. american fuzzy lop (2.35b). <http://lcamtuf.coredump.cx/afl/>. Accessed December 20, 2016.

# ProFL: A Fault Localization Framework for Prolog

George Thompson

North Carolina A&T State University, USA

gmthompson1@aggies.ncat.edu

Allison K. Sullivan

The University of Texas at Arlington, USA

allison.sullivan@uta.edu

## ABSTRACT

Prolog is a declarative, first-order logic that has been used in a variety of domains to implement heavily rules-based systems. However, it is challenging to write a Prolog program correctly. Fortunately, the SWI-Prolog environment supports a unit testing framework, plunit, which enables developers to systematically check for correctness. However, knowing a program is faulty is just the first step. The developer then needs to fix the program which means the developer needs to determine what part of the program is faulty. ProFL is a fault localization tool that adapts imperative-based fault localization techniques to Prolog's declarative environment. ProFL takes as input a faulty Prolog program and a plunit test suite. Then, ProFL performs fault localization and returns a list of suspicious program clauses to the user. Our toolset encompasses two different techniques: ProFL<sub>s</sub>, a spectrum-based technique, and ProFL<sub>m</sub>, a mutation-based technique. This paper describes our Python implementation of ProFL, which is a command-line tool, released as an open-source project on GitHub (<https://github.com/geoorge1d127/ProFL>). Our experimental results show ProFL is accurate at localizing faults in our benchmark programs.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Prolog, Fault localization, Declarative programming

### ACM Reference Format:

George Thompson and Allison K. Sullivan. 2020. ProFL: A Fault Localization Framework for Prolog. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404367>

## 1 INTRODUCTION

Declarative programming languages enable developers to implement rule-based systems, e.g. fault tolerant software defined network layers [10], and reason over the designs of systems, e.g. disproving the Chord ring-maintenance protocol [14]. Declarative languages express the logic of a system's behavior without giving any control flow. Then, the execution determines how to achieve

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404367>

the desired behavior. While a number of declarative languages exist [4, 6, 9], the broad adoption of these languages is often hindered for two key reasons. First, declarative languages have steep learning curves since their execution environment is inherently different than commonly used imperative languages, e.g. Java and C++, which use statements to sequentially change a program's state into the desired behavior. Second, declarative languages often lack robust tool support for test automation frameworks.

Prolog is a first-order logic based declarative language that has seen a wave of new users due to its applicability to rising fields such as machine learning [2]. While Prolog is viewed as one of the go to languages for artificial intelligence systems, several studies have shown that Prolog is difficult for many programmers to learn [11, 13]. Fortunately, SWI-Prolog [12] contains a unit testing framework, plunit. Besides support for unit testing, plunit is actively developing frameworks to automatically generate tests and calculate coverage, which help developers reveal bugs. However, just finding bugs is not sufficient. The user needs to be able to fix the bug. Unfortunately, locating the faulty portion of a Prolog program is difficult. Imperative languages' sequential execution allows developers to step over changes to a program's state to locate where the execution goes wrong. This is not possible in Prolog, which has no notion of control flow. Furthermore, Prolog programs can have numerous interdependent clauses spanning multiple non-contiguous lines, making the process of locating bugs time consuming.

This paper describes our Python implementation of ProFL, a toolset to perform automated fault localization for Prolog. ProFL is a command line tool that we release as an open-source project (<https://github.com/geoorge1d127/ProFL>). ProFL takes as input a faulty Prolog program and a corresponding plunit test suite. ProFL then provides two different types of fault localization techniques: spectrum-based techniques, which are based on well-established fault localization techniques for imperative languages [1, 5, 8], and a mutation-based technique, which is based on a recent advancement in fault localization [7]. Lastly, ProFL visualizes the results back to the user to help steam-line debugging.

## 2 BACKGROUND

ProFL integrates with SWI-Prolog, which is currently the most commonly used compilation environment for Prolog programs [12]. Figure 1 depicts a real world faulty Prolog program<sup>1</sup>. Prolog programs are a collection of clauses that create a knowledge base that can be queried and tested. Clauses can either be a fact or a rule and are always terminated by a period. Lines 1 - 6 depict *facts*, which use a predicate expression to make a declarative statement about the problem domain. Line 8 depicts a *rule*, which uses a predicate expression to describe relationships among facts. A rule “A :- B” can read as “A if B” where “:-” represents logical implication. Thus,

<sup>1</sup><https://stackoverflow.com/questions/49353041/prolog-query-satisfiable-but-returns-false>

```

1. male(william). /*william is male.*/
2. male(harry).
3. parent(william, diana). /*parent(x,y) - the parent of x is y.*/
4. parent(william, charles).
5. parent(harry, diana).
6. parent(harry, charles).
7. /*brother(X,Y) - the brother of X is Y.*/
8. brother(X,Y) :- X\=Y, parent(X,A), parent(Y,A), male(Y).

```

Figure 1: Faulty Family Tree Prolog Program

rules express ways to derive or compute new facts. On line 8, the left-hand side of the rule defines the predicate brother which takes as arguments two variables X and Y. The right-hand side of the rule uses negation ( $\neq$ ), unity ( $=$ ), conjunction ( $\wedge$ ) and predicates (parent, male) to express that Y is X's brother if all of the following holds: (1) X cannot unify to Y, (2) there is a variable A that is a parent to both X and Y, and (3) Y is male.

Rather than sequentially executing statements, Prolog's execution environment tries to prove that a query, a user-specified sequence of predicates, is true using the defined facts and rules. plunit, Prolog's unit testing library, defines a unit test to be Prolog queries that are expected to be true for a given program. Below is a set of tests which reveal the fault on line 8 in Figure 1:

```

1. :- begin_tests(family).
2. :- include(family).
3. test(test1) :- brother(william,harry). /*Test Passes.*/
4. test(test2) :- brother(william,X). /*Test Fails.*/
5. :- end_tests(family).

```

The error arises because the negation of unity (" $\neq$ ") is used as if it asserts the two variables must be different, which is not how the constraint gets interpreted. When Prolog resolves " $X \neq Y$ " the execution environment asks "Can X and Y ever be unified? If so, fail." This incorrect usage does not impact test1's execution as both arguments to brother are bound to different constants and therefore cannot unify. In contrast, for test2, one of the arguments is a constant (william) and the other is an unbound variable (X). Since none of the clauses prevent X from being bound to william, the two can unify resulting in test2 failing. What the user really wanted to constrain is "in any solution, X and Y must be different." In Prolog, this can be achieved by making the following change:

```
8. brother(X,Y) :- dif(X,Y), parent(X,A), parent(Y,A), male(Y).
```

The predicate `dif` is a constraint that is true if and only if X and Y are different terms.

### 3 TECHNIQUE

ProFL adapts well-studied imperative fault localization techniques to Prolog's declarative environment. Figure 2 gives an overview of ProFL's components. ProFL takes as input (1) a faulty Prolog program and (2) a plunit test suite. Then, depending on the user's selection, ProFL will run ProFL<sub>s</sub>, which performs spectrum-based fault localization (SBFL), and/or ProFL<sub>m</sub>, which performs mutation-based fault localization (MBFL). As output, for each fault localization technique run, ProFL displays the likely faulty clauses from most to least suspicious in separate tables. We next describe how the four main components of ProFL work.

#### 3.1 Coverage Engine

Both SBFL and MBFL rely on accurate coverage information to be effective. In Prolog's execution, all clauses form one knowledge

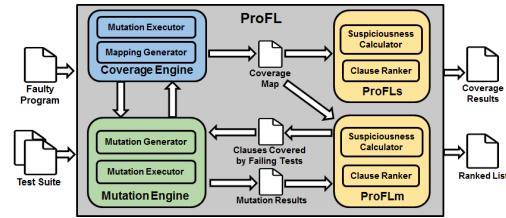


Figure 2: ProFL Component Diagram

base that is leveraged to try and prove a query true. To provide effective localization, we want to consider a clause covered by a test if that clause is *actively* used to resolve the test's query. Therefore, we want something analogous to statement coverage for imperative programs. Unfortunately, plunit's coverage framework is under development and does not provide the fine-grained, clause level data ProFL needs. Therefore, ProFL implements a recently developed coverage technique which utilizes mutants to calculate clause coverage [3]. To start, the mutation engine (Section 3.2) is invoked to produce all valid mutants of the program. Then, the coverage engine performs a streamlined version of mutation testing. For every mutant  $m$ , the test suite is executed. If a test  $t$  kills  $m$ , then  $t$ 's pass/fail result changed compared to the original program. Since  $m$  is a mutation of some clause  $c$ , we conclude  $t$  covers  $c$ . As an optimization, once the coverage engine has detected  $t$  covers  $c$ ,  $t$  is not executed for any remaining mutants of  $c$ . Using this process, the coverage engine builds a map from every clause to the test case(s) that cover it. This map is then passed as input to both ProFL<sub>s</sub> and ProFL<sub>m</sub>. The coverage engine is intentionally kept modular to allow for the integration of other suitable coverage techniques.

#### 3.2 Mutation Engine

Our mutation engine implements the following mutation operators: removing predicates, interchanging disjunction and conjunction, changing atoms or variables to anonymous variables, and interchanging arithmetic operators. When mutating a Prolog program, it is possible to accidentally create non-deterministic predicates which leads to an infinite loop during Prolog's execution. Therefore, we follow the results from a recent empirical study and avoid two types of mutants, clause reversal and cut transformation, known to create non-deterministic predicates [3]. For the coverage engine, the mutation engine generates all possible non-equivalent mutants for each clause in the program but does not control execution. For ProFL<sub>m</sub>, the mutation engine generates all possible non-equivalent mutants for each clause covered by some failing test. Then, the mutation engine executes each mutant  $m$  against the original test suite. As it executes, the mutation engine builds a mapping from  $m$  to the tests that pass  $m$  and the tests that fail  $m$ . Unlike the coverage engine, the mutation engine always executes the entire test suite against each mutant, as ProFL<sub>m</sub> needs detailed information about how test results change from the original to the mutated program.

#### 3.3 ProFL<sub>s</sub>: Spectrum-Based Fault Localization

ProFL allows users to perform traditional SBFL [1, 5, 8]. To evaluate how likely a program statement is to be faulty, SBFL utilizes test cases' pass/fail results and information about what portions of the program those test cases execute to calculate a suspiciousness

Name	Formula
tarantula [5]	$\frac{\frac{failed(c)}{totalfailed}}{\frac{failed(c)}{totalfailed} + \frac{passed(c)}{totalpassed}}$
ochiai [1]	$\frac{failed(c)}{\sqrt{totalfailed \times (failed(c) + passed(c))}}$
op2 [8]	$failed(c) - \frac{passed(c)}{totalpassed+1}$

*totalfailed*: total number of test cases that failed.  
*totalpassed*: total number of test cases that pass.  
*failed(c)*: number of failed test cases that cover *c*.  
*passed(c)*: number of passed test cases that cover *c*.

**Figure 3: ProFL<sub>s</sub> Supported Suspiciousness Formulas**

score for each statement. The higher a statement's suspiciousness score, the more likely the statement is to be faulty. To perform SBFL, ProFL<sub>s</sub> uses the initial test results and the coverage mapping from Section 3.1 to calculate the suspiciousness score of each clause using the three formulas outlined in Figure 3. These three formulas are commonly used in imperative SBFL techniques. *totalfailed* and *totalpassed* are the number of failed and passed test cases for the Prolog program. *failed(c)* and *passed(c)* are the number of failed and passed tests that utilize clause *c*, which is determined by our coverage framework. In general, these formulas are designed to assign a higher suspiciousness score to a clause that is covered by more failing tests and fewer, if any, passing tests. After all suspiciousness scores are calculated, ProFL<sub>s</sub> creates a ranked list of clauses from most suspicious to least, which gets displayed to the end user.

### 3.4 ProFL<sub>m</sub>: Mutation-Based Fault Localization

MBFL techniques rely on tactically altering program statements and seeing how these changes affect the test results [7]. If a mutation causes a failing test to pass, then the mutated location becomes more suspicious. However, if a mutation causes a passing test to fail, then the mutated location becomes less suspicious. To perform MBFL, ProFL<sub>m</sub> extracts the clauses that are covered by failing tests from the coverage map in Section 3.1. Then, ProFL<sub>m</sub> uses the mutation engine from Section 3.2 to get a map between mutants of these clauses and which tests pass and fail each mutant. Rather than using the mutation testing information to build a mutation score for the test suite, ProFL<sub>m</sub> uses the information to help calculate a suspiciousness score for each clause using the formula in Figure 4. Within the summation, the first term correlates to the likelihood of *c* being faulty, as the term increases when mutating *c* causes a failing test to pass. The second term correlates to the likelihood of *c* not being faulty, as the term decreases when mutating *c* causes a passing test to fail. The weight  $\alpha$  is based on the number of test result changes and is used to help balance the two terms, since breaking a program is easier than correcting a program. Similar to ProFL<sub>s</sub>, ProFL<sub>m</sub> calculates a suspiciousness score for each clause and then sends a ranked list of suspicious clauses to be formatted and displayed to the end user.

## 4 USAGE

In this section, we describe how users can invoke ProFL. More details can be found on the ProFL GitHub homepage.

Muse Formula [7]
$\frac{1}{ mut(c) } \sum_{m \in mut(c)} \left( \frac{ f_P(c) \cap p_m }{ f_P } - \alpha * \frac{ p_P(c) \cap f_m }{ p_P } \right)$ where $\alpha = \frac{f2p}{ mut(P)  \cdot  f_P } \cdot \frac{ mut(P)  \cdot  p_P }{p2f}$

*f<sub>P</sub>(c)*: set of test cases that cover *c* and fail *P*.  
*p<sub>P</sub>(c)*: set of test cases that cover *c* and pass *P*.  
*mut(c)*: set of all mutants of *P* that mutate *c* and change a test result.  
*f<sub>m</sub>*: set of failing test for mutant *m*.  
*p<sub>m</sub>*: set of passing test for mutant *m*.  
*mut(P)*: number of mutants created from program *P*  
*f2p*: number of tests that change from failing to passing.  
*p2f*: number of tests that change from passing to failing.

**Figure 4: ProFL<sub>m</sub> Supported Suspiciousness Formula**

To localize a fault, run: `python ProFL.py -p <arg> -t <arg> -f <arg> -v <arg> [-s <arg>] [-r <arg>] [-c <arg>]` or `python ProFL.py --program-path <arg> --test-suite <arg> --fl-technique <arg> --view <arg> [--suspicious-formula <arg>] [--result-path <arg>] [--coverage-path <arg>]`

- `-p, --program-path`: This argument is *required*. Pass the file name of the faulty Prolog program.
- `-t, --test-suite`: This argument is *required*. Pass the file name of the plunit test suite.
- `-f, --fl-technique`: This argument is *required*. Pass the fault localization technique to use. The value should be `"spectrum"`, `"mutation"`, or `"both"`.
- `-v, --view`: This argument is *required*. Pass how much of the ranked suspicious list to view. The value should be `"top1"`, `"top5"`, `"top10"`, or `"all"`.
- `-s, --suspicious-formula`: This argument is *optional* and is used when the technique is `"spectrum"` or `"both"`. Pass the suspiciousness formula for ProFL<sub>s</sub> to use. The value should be `"tarantula"`, `"ochiai"`, or `"op2"`. If specifying more than one, separate with a comma. If not specified, all three are used.
- `-r, --result-path`: This argument is *optional*. Pass the path to which you want to save the fault localization results. If not specified, the results are only printed to the terminal.
- `-c, --coverage-path`: This argument is *optional*. Pass the path to which you want to save the coverage results. If not specified, the coverage information is not saved.

The ProFL tool reports one table per suspiciousness formula executed. Depending on the user's selection, this table will display the most suspicious clause, the top 5 suspicious clauses, the top 10 suspicious clauses, or all ranked clauses. For each suspicious clause *c*, ProFL reports: (1) the number of test that *c* fails, (2) the number of test that *c* passes, and (3) the suspiciousness score of *c*. The user can also elect to save the fault localization and coverage results.

## 5 EVALUATION

Table 1 shows the 10 Prolog programs used to evaluate ProFL and the corresponding performance of ProFL. These programs are incorrect submissions to Exercism's Prolog exercises track<sup>2</sup>. Program

<sup>2</sup><https://exercism.io/tracks/prolog/exercises>

**Table 1: ProFL Performance Results**

Program	Program Details			Time (s)		Ranking			
	# Pred	# Cls	# Tests	ProFL <sub>s</sub>	ProFL <sub>m</sub>	Tarantula	Ochiai	Op2	Muse
anagram	6	7	18	83.2	89.9	1	1	1	1
binary	5	3	16	N/A	N/A	N/A	N/A	N/A	N/A
complex_num	8	8	27	239.6	246.5	1	1	1	2
dominoes	5	7	12	39.8	69.0	1	1	1	2
grains	2	5	12	49.8	66.9	2	1	1	1
hamming	4	3	15	33.1	48.5	1	1	1	2
pascal_tri	6	6	8	23.1	30.3	1	1	2	6
queen_attack	5	5	13	68.3	80.9	1	1	1	2
space_age	3	10	8	23.0	32.2	9	9	9	1
triangle	2	6	19	84.4	143.9	1	1	1	2

is the name of the program: **anagram** determines which words are anagrams of a given word, **binary** converts a binary number to a decimal number, **complex\_num** implements complex numbers, **dominoes** makes a chain of dominoes, **grains** calculates the number of grains of wheat on a chessboard when the number on each square doubles, **hamming** calculates the Hamming difference between two DNA strands, **pascal\_tri** computes Pascal’s triangle, **queen\_attack** determines if two queens can attack each other on a chess board, **space\_age** calculates how old someone is in terms of a planet’s solar years, and **triangle** determines the type of a triangle.

The next three columns in Table 1 describe the size and complexity of the programs. **# Pred** is the number of unique predicates, **# Cls** is the number of clauses and **# Tests** is the size of the plunit test suite. We use the test suites released by Exercism for each exercise. The next two columns represent the runtime for ProFL<sub>s</sub> and ProFL<sub>m</sub> respectively. The times are in seconds and captures the time it takes to go from processing the input to displaying the results to the user. The last four columns represents the ranking given to the faulty clause by each suspiciousness formula support by ProFL: **Tarantula**, **Ochiai** and **Op2** for ProFL<sub>s</sub> and **Muse** for ProFL<sub>m</sub>. The lower this value is, the better the fault localization technique performed. A value of 1 means the faulty statement was ranked as the most suspicious clause while a value of 6 means the faulty clause was ranked as the 6th most suspicious clause.

For one program, **binary**, ProFL is unable to perform fault localization as our technique got stuck in an infinite loop when mutating the faulty statement during the initial coverage calculations. This motivates our goal to incorporate additional coverage options in future releases of the tool. For the remaining 9 models, the best performing results are presented in green for both runtime and ranking. Across all programs, ProFL<sub>s</sub> runs faster then ProFL<sub>m</sub> for an average speed up of 1.4x. This is expected as ProFL<sub>m</sub>’s suspiciousness formula requires targeted mutation testing in addition to the initial coverage results while ProFL<sub>s</sub>’s suspiciousness formula works just with the initial coverage results. For ProFL<sub>s</sub>, all three suspiciousness formulas have comparable performance with Ochiai marginally performing the best. In comparison, ProFL<sub>m</sub>’s Muse formula performs slightly worse than the ProFL<sub>s</sub>’s formulas, often ranking the faulty statement in an equal position (3 of 9) or one position lower (5 of 9). However, for **space\_age**, ProFL<sub>m</sub> significantly outperforms ProFL<sub>s</sub>. For this model, the faulty clause causes almost all tests to fail, impeding ProFL<sub>s</sub>’s performance. This highlights that

ProFL<sub>m</sub> can perform well in situations where ProFL<sub>s</sub> is unable to be effective. Therefore, given that the same initial coverage information can be re-used between ProFL<sub>s</sub> and ProFL<sub>m</sub>, user can increase their confidence by running both techniques and leverage their individual strengths, while incurring the modest overhead from the additional mutation testing needed for ProFL<sub>m</sub>. Overall, these results show ProFL can help locate faults and our two techniques have complementary roles to each other.

## 6 CONCLUSION

This paper introduced the open-source ProFL tool for automated fault localization of Prolog programs. ProFL provides command-line options to automatically perform any combination of spectrum-based or mutation-based fault localization. Given a faulty Prolog program and a fault-revealing test suite, ProFL is able to report a list of suspicious clauses for the user to investigate. Our experiments show ProFL has a minor overhead and promising accuracy results.

## REFERENCES

- [1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *JSS* (2009).
- [2] Ivan Bratko. 2001. *Prolog programming for artificial intelligence*. Pearson education.
- [3] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. 2018. Measuring coverage of prolog programs using mutation testing. In *International Workshop on Functional and Constraint Logic Programming*. Springer, 39–55.
- [4] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM TOSEM* (2002).
- [5] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *ASE*.
- [6] Simon Marlow et al. 2010. Haskell 2010 language report. Available online [http://www.haskell.org/\(May 2011\)](http://www.haskell.org/(May 2011)) (2010).
- [7] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST*.
- [8] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *TSE* (2011).
- [9] Philip J. Pratt and Mary Z. Last. 2008. *A Guide to SQL* (8th ed.). Course Technology Press.
- [10] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: Declarative Fault Tolerance for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. 109–114.
- [11] Maarten W. Van Someren. 1990. What’s wrong? Understanding beginners’ problems with Prolog. *Instructional Science* 19, 4/5 (1990), 257–282.
- [12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.
- [13] Shanshan Yang and Mike Joy. 2007. Approaches for Learning Prolog Programming. *Innovation in Teaching and Learning in Information and Computer Sciences* 6, 4 (2007), 88–107.
- [14] Pamela Zave. 2012. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.* 42 (2012), 49–57.

# FineLock: Automatically Refactoring Coarse-Grained Locks into Fine-Grained Locks

Yang Zhang

School of Information Science and Engineering,  
Hebei University of Science and Technology  
Shijiazhuang, China  
zhangyang@hebust.edu.cn

Juan Zhai

Rutgers University  
Piscataway, USA  
juan.zhai@rutgers.edu

Shuai Shao

School of Information Science and Engineering,  
Hebei University of Science and Technology  
Shijiazhuang, China  
shao724854691@163.com

Shiqing Ma

Rutgers University  
Piscataway, USA  
shiqing.ma@rutgers.edu

## ABSTRACT

Lock is a frequently-used synchronization mechanism to enforce exclusive access to a shared resource. However, lock-based concurrent programs are susceptible to lock contention, which leads to low performance and poor scalability. Furthermore, inappropriate granularity of a lock makes lock contention even worse. Compared to coarse-grained lock, fine-grained lock can mitigate lock contention but difficult to use. Converting coarse-grained lock into fine-grained lock manually is not only error-prone and tedious, but also requires a lot of expertise. In this paper, we propose to leverage program analysis techniques and pushdown automaton to automatically covert coarse-grained locks into fine-grained locks to reduce lock contention. We developed a prototype *FineLock* and evaluates it on 5 projects. The evaluation results demonstrate *FineLock* can refactor 1,546 locks in an average of 27.6 seconds, including converting 129 coarse-grained locks into fine-grained locks and 1,417 coarse-grained locks into read/write locks. By automatically providing potential refactoring recommendations, our tool saves a lot of efforts for developers.

## CCS CONCEPTS

- Software and its engineering → Software evolution.

## KEYWORDS

Refactoring, Fine-grained lock, Static analysis, Read-write lock, Pushdown automaton

### ACM Reference Format:

Yang Zhang, Shuai Shao, Juan Zhai, and Shiqing Ma. 2020. FineLock: Automatically Refactoring Coarse-Grained Locks into Fine-Grained Locks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404368>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 978-1-4503-8008-9/20/07...\$15.00  
<https://doi.org/10.1145/3395363.3404368>

## 1 INTRODUCTION

Lock is frequently used to guarantee the correctness of shared resources access in concurrent programs. To enforce exclusive access to a shared resource, only one thread can acquire the lock of a given resource at a time while other threads have to wait for the thread to release the lock. Unfortunately, concurrent programs based on locks are susceptible to lock contention which would result in low performance and poor scalability. To avoid lock contention, software transactional memory and lock-free algorithm have been proposed to control access to shared memory in concurrent programs. However, lock is still one of the most popular synchronization mechanisms among developers, which motivates us to provide automated support in refactoring lock-related code to reduce lock contention.

Inappropriate granularity of a lock makes lock contention even worse. It is generally accepted that employing a coarse-grained lock may exacerbate lock contention while a fine-grained lock may mitigate lock contention by decreasing the waiting time of other threads that attempt to acquire the lock. Consequently, there is a strong need to employ fine-grained lock. However, writing a program based on fine-grained lock is much more difficult compared with that based on a coarse-grained lock, which requires careful design and expertise. For example, developers have to consider which code patterns are applicable for fine-grained locks. Furthermore, when developers are required to maintain existing code that have coarse-grained locks, it would be error-prone and tedious to convert existing coarse-grained locks to fine-grained ones manually. This inspires us to develop an automation tool to help developers refactor code by converting coarse-grained locks into find-grained lock with the hope of reducing lock contention.

Some existing tools have been proposed to refactor locks automatically. McCloskey et al. [8] presented an automated tool *Autolocker* to convert pessimistic atomic sections into standard lock-based code. Schäfer et al. [10] presented a refactoring tool *Relocker* to convert a synchronized lock into a *ReentrantLock* and a *ReentrantReadWriteLock*. Both *Autolocker* and *Relocker* conducted refactoring for coarse-grained locks. Moreover, *Relocker* takes the critical section as a whole to analyze the read/write operations. Without performing detailed analysis for every operation in a critical section, *Relocker* is incapable of converting locks into fine-grained locks, which is confirmed by experiments on several benchmarks.

Zhang et al. [12] presented an automated refactoring tool *CLOCK* to convert a synchronized lock into a *StampedLock*. *CLOCK* could not only transform read and write locks, but also refactor downgrading/upgrading and optimistic read locks. However, *CLOCK* could conduct limited refactoring due to the non-reentrancy of *StampedLock*. Some commercial refactoring tools, such as concurrency-oriented refactoring for JDT [9] and LockSmith [11], had been integrated into Eclipse and IntelliJ to merge locks, convert them, and make the field atomic. However, none of the existing tools is able to downgrade a coarse-grained synchronized lock into a fine-grained read-write lock.

Automatically refactoring a coarse-grained lock into a fine-grained read-write lock faces a few challenges. Firstly, automatically inferring an appropriate fine-grained lock for a critical section is extremely difficult, even for humans, which requires expertise. For example, a downgrading lock firstly uses a write lock and then employs a read lock. This entails us to automatically identify read operations and write operations from source code. Secondly, automatically refactoring into fine-grained lock requires to modify existing code without introducing new bugs. For example, it needs to locate a right position to create a lock object and ensure the lock is used correctly.

In this paper, we propose a novel solution for converting a coarse-grained synchronized lock into a fine-grained *ReentrantReadWriteLock* lock and develop a prototype named *FineLock*. Firstly, we leverage program analysis techniques to identify critical sections that can be refactored. Then we automatically identify read statements and write statements in the critical sections, and summarize them as a pattern (a sequence of characters where each character indicates whether a statement is a *read* operation or a *write* operation). After that, we feed the pattern into a pushdown automaton to decide what kind of refactor should be performed. Finally, based on the result from the automaton, a coarse-grained lock is transformed into a fine-grained lock including a downgrading lock and a splitting lock. We develop a prototype *FineLock* as an Eclipse plugin. We evaluate it on 5 real-world projects. The results show *FineLock* refactors 1,546 locks in an average of 27.6 seconds, including converting 129 coarse-grained locks into fine-grained locks and 1,417 coarse-grained locks into read/write locks, which improves throughput for these projects.

## 2 MOTIVATION

We showcase how our approach can reduce lock contention using the examples in Figure 1. Figure 1(a) shows a typical implementation of cache processing where the method is protected by a synchronized lock (line 1). It first checks whether the cache contains data in line 2. If so, the data is read (line 5). Otherwise, data will be written into the cache (line 3). Figure 1(b) presents the code refactored by *Relocker* [10]. According to the lock inference strategy of *Relocker*, a coarse-grained write lock is inferred (lines 22 and 29). However, the write operation is executed only when the cache does not have the data. The refactored code by *Relocker* is still too coarse-grained and has low concurrency. To allow more concurrency and reduce lock contention, we propose our approach to infer fine-grained locks. For the code in Figure 1(a), a read lock is inferred using our approach and the result code is shown in Figure 1(c). It first acquires

```

1 public synchronized void cached() {
2     if (!cacheValid) {
3         ... // write into cache
4     }
5     ... // read data
6 }
7
8 (a)
9
10 public void cached() {
11     lock.writeLock().lock();
12     try {
13         if (!cacheValid) {
14             ... // write into cache
15             lock.readLock().lock();
16         }
17         ... // read data
18     } finally {
19         lock.writeLock().unlock();
20     }
21 }
22
23 (b)
24
25 public void cached() {
26     lock.readLock().lock();
27     try {
28         if (!cacheValid) {
29             ... // write into cache
30             lock.writeLock().lock();
31         }
32         ... // read data
33     } finally {
34         lock.readLock().unlock();
35     }
36 }
37
38 (c)
39
40 public void cached() {
41     lock.readLock().lock();
42     if (!cacheValid) {
43         lock.readLock().unlock();
44         lock.writeLock().lock();
45     }
46     try {
47         if (!cacheValid) {
48             ... // write into cache
49             lock.readLock().lock();
50         }
51     } finally {
52         lock.writeLock().unlock();
53     }
54 }
55
56 public void cached() {
57     lock.readLock().lock();
58     try {
59         ... // read data
60     } finally {
61         lock.readLock().unlock();
62     }
63 }
64
65 (c)

```

Figure 1: The motivating example

a read lock (line 42) which is enough to ensure concurrency for the condition checking in line 43. When the condition holds, a written lock must be used and thus the read lock is released (line 44) and a write lock is acquired (line 45). Note that the condition is rechecked in line 47 to guarantee the consistency since the condition might be changed by another thread. After the data is written to the cache (line 48), the write lock is downgraded into a read lock (line 52) to allow more concurrency. Finally, after the data is read (line 56), the read lock is released in line 58. The example demonstrates that our tool can reduce lock contention compared to existing tools by converting coarse-grained locks into fine-grained locks.

## 3 DESIGN

Figure 2 gives the design of our approach. *FineLock* takes the source code based on coarse-grained locks as an input. It firstly analyzes each critical section by visitor pattern analysis, and then collects synchronized methods and blocks, as well as monitor objects. *FineLock* judges whether monitor objects are aliased or not. Based on results of alias analysis, it can generate a *ReentrantReadWriteLock* in each class. Secondly, read/write sequence of each critical section is obtained by side effect analysis. And then a pushdown automaton is defined to identify and to accept these read-write sequence. Both read/write lock and fine-grained lock including downgrading lock and splitting lock are recommended for each critical section. Finally, based on these analysis results, *FineLock* convert each synchronized method and block into fine-grained *ReentrantReadWriteLock*.

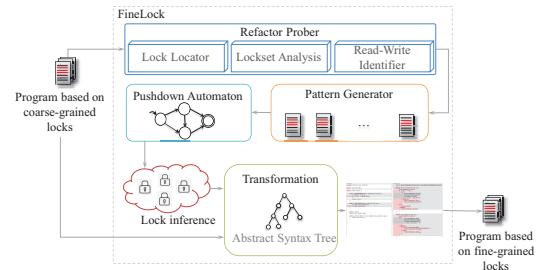


Figure 2: Overview of *FineLock*

### 3.1 Refactor Prober

Refactor prober is to locate synchronized locks that can be refactored into fine-grained locks and summarize the statements in a

critical section into a pattern which will be used by pushdown automaton to infer locks. Here a pattern means a sequence of characters where each character indicates whether a statement is a read operation or a write operation.

**Lock Locator.** Given a project, we first generate an abstract syntax tree (AST) using Eclipse JDT [4] for each source file and then leverage visitor pattern to locate all synchronized methods/blocks with their monitor objects by traversing AST.

**Lockset Analysis.** *FineLock* constructs a hash set *LockSet* in which a monitor object is the key and a *ReentrantReadWriteLock* object is the value. We leverage Wala [6] to perform alias analysis for each monitor object since the same monitor object in the original program should use the same *ReentrantReadWriteLock* instance in the refactored program.

**Read-Write Identifier.** For each statement in a critical section, the read-write identifier would identify it as a read operation or a write operation by conducting side-effect analysis. If a write lock is used to synchronize a read operation, it would downgrade the performance. This entails us to use an appropriate lock that is enough to ensure the correctness of concurrent programs, which means a statement should be identified as read or write. In *FineLock*, the following statements are identified as write operations: 1) a statement that modifies a static field or variant, 2) a method invocation whose callee contains a write operation, and 3) a library method whose code is unavailable or cannot be determined.

**Pattern Generator.** The pattern generator is to encode the sequence of programming statements into a sequence of characters where each character marks the state of the statement. Each character sequence is called a pattern and it would be feed into automaton to infer an appropriate lock. An identified read operation is marked as *r* and an identified write statement is marked as *w*. Also, *FineLock* marks *if* statements since we need to know the read/write property of the condition checking expression in the *if* statement to infer locks. The beginning of an *if* statement is marked as *c* and the end of an *if* statement is marked as *e*.

### 3.2 Pushdown Automaton

The pushdown automaton is utilized to infer locks based on a generated pattern for a critical section. Each character in the pattern sequence is used as a trigger to transfer from one state to another. The state transition diagram is presented in Figure 3.

The pushdown automaton  $M_{fg} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  is a seven-tuple.  $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$  is a finite set of states and  $q_0$  is the start state.  $\Sigma = \{r, w, c, e\}$  is the input alphabet (defined in Section 3.1).  $\Gamma = \{Z_0, V, C, D, A, B\}$  is the stack alphabet, and each value indicates the current status of the inferring process.  $Z_0$  indicates an empty stack.  $V$  indicates halting the automaton, which means the given pattern does not meet any refactoring condition, and thus a write lock is inferred for the sake of safety.  $C, D, A$ , and  $B$  separately indicate the status based on the traversed sequence, where  $C$  indicates the traversed sequence is in an *if* statement,  $D$  indicates the traversed sequence contains a downgrading lock.  $A$  and  $B$  indicate splitting locks. Specifically,  $A$  means a read lock followed by a write lock and  $B$  means a write lock followed by a read lock. The state transition  $\delta$  is a mapping set  $Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^+$  where  $\Gamma^+$  is the positive closure of  $\Gamma$ .  $\delta$  is defined as  $\langle q, x, X, q', T \rangle$

where  $q$  is a state,  $x$  is an element of  $\Sigma$ ,  $X$  is a stack symbol and  $T$  is a stack operation. It means if the current state is  $q$ , the input is  $x$  and the top element of the stack is  $X$ , we would transfer to state  $q'$ , and execute stack operation  $T$ . The stack operation  $T$  consists of  $X$ ,  $X'X$ , and  $\rho$ , where  $X$  indicates transition without stack operation,  $X'X$  indicates pushing  $X'$  into the stack, and  $\rho$  indicates popping the top element of the stack. To simplify the representation, we use  $\langle x, X/T \rangle$  to represent for the transition  $\langle q, x, X, q', T \rangle$ .  $F$  is a finite set of final states and  $F = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ , where  $q_1$  and  $q_2$  indicates that a read lock and a write lock is inferred respectively,  $q_3, q_4$  and  $q_5$  indicate a splitting lock is inferred, and  $q_6$  indicates a downgrading lock is inferred.

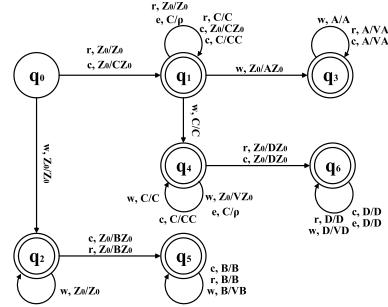


Figure 3: Pushdown Automaton

**Examples.** Take the code in Figure 1(a) as an example. The pattern sequence “*cwer*” is generated for the method *cached()* in line 1 and it is used as the input for pushdown automaton to infer refactored locks. To start with, the stack is initialized as empty using the symbol  $Z_0$ . The first character of the given pattern is *c*, which indicates a condition evaluation of an *if* statement, and thus we transfer to state  $q_1$  and push symbol *C* into the stack. Given the next character *w* and the top symbol *C*, it implies that the write operation is in an *if* statement, and we can transfer from state  $q_1$  to state  $q_4$ . In the next pass, we have *e* as the input and the top symbol of the stack is *C*, which indicates the end of the *if* statement. Hence we pop the top symbol *C* from the stack. The last character is *r* and the top element in the stack is  $Z_0$ , it indicates a read operation after a *if* statement, and thus we infer *lock downgrading* and push *D* into the stack. All the characters in the pattern have been visited and we are in the final state  $q_6$ , meaning the pattern is accepted by the automaton and a downgrading lock is inferred for later refactoring.

### 3.3 Transformation

*FineLock* conducts transformation on AST by converting synchronized locks into *ReentrantReadWriteLock* locks. In each refactored class, *FineLock* first imports the *ReentrantReadWriteLock* package and then defines the instance variable with the type *ReentrantReadWriteLock*. All locks are put into the *try..finally..* structure to ensure releasing a lock in case an exception is triggered. For downgrading lock, *FineLock* handles the transformation in the same way as the code shown in Figure 1(c). For splitting lock, *FineLock* performs the splitting operations for each read/write operation.

## 4 EVALUATION

We implement a prototype *FineLock* leveraging Wala [6], and the prototype is integrated as a plug-in of Eclipse, and we empirically evaluate it to address the following question:

**RQ1:** How effective and efficient is *FineLock* in refactoring coarse-grained locks into fine-grained locks?

**RQ2:** How useful is *FineLock* in improving throughput?

The evaluation was conducted on a HP Z240 workstation with 3.6 GHz Intel Core i7 CPU and 8GB main memory. The operating system is Ubuntu 16.04, and the JDK version is 8.

### 4.1 Effectiveness and Efficiency in Refactoring

We evaluate the effectiveness and efficiency of *FineLock* in five real-world applications including HSQLDB [5], Jenkins [7], Cassandra [1], JGroups [2], and SPECjbb2005 [3]. The size of the projects varies from 12,519 to 431,022 source lines of code (SLOC), and the time used to refactor each project is on average 27.6s, which clearly indicates our prototype is generally applicable to large projects.

The evaluation results are summarized in Table 1, which presents the projects (column 1), the numbers of synchronized locks in the original projects (column 2) and the data after refactoring (columns 3-9). Each synchronized lock in the original projects is refactored into one of the following locks: a downgrading lock (column 3), a splitting lock (column 4), a read lock (column 5) or a write lock (column 6). Downgrading locks and splitting locks compose fine-grained locks while read locks and write locks compose *ReentrantReadWriteLock* locks. Column 7 shows the ratio between the total number of fine-grained locks (columns 3-4) and the total number of synchronized locks (column 2) and column 8 shows the ratio between the total number of *ReentrantReadWriteLock* locks (columns 5-6) and the total number of synchronized locks (column 2). The last column gives the ratio between the total number of refactored locks and the total number of synchronized locks, which demonstrates that our tool can achieve 100% lock refactoring.

Table 1: Refactored Locks by *FineLock*

Benchmark	Original		Refactored Locks					
	#LOCKS	#DL	#SL	#RL	#WL	%FINE	%R/W	%RE
HSQLDB	684	6	39	109	530	7%	93%	100%
Jenkins	274	3	14	19	238	6%	94%	100%
Cassandra	239	2	24	39	174	11%	89%	100%
JGroups	179	5	33	28	113	21%	79%	100%
SPECjbb2005	170	1	2	38	129	2%	98%	100%
Total	1,546	17	112	233	1,184	8%	92%	100%

To further answer the question, we check whether the code after refactoring by *FineLock* still have the same behaviors as the original code by running test cases and manually checking. Specifically, we run the existing developer test cases of all 5 projects, and the percentage of passed test cases is 100%. In addition, we manually check all refactored locks. Each refactored lock is assigned to two developers to avoid bias. In the process, we manually check the following aspects: 1) if the refactoring changes the behaviors of original code; 2) if an inferred lock is correct; 3) if a lock is inserted to the right position; and 4) if a lock is used correctly. The manual results demonstrate that all our refactored locks are correct.

### 4.2 Improving Throughput

To answer RQ2, we run the concurrency test cases of the two projects HSQLDB and SPECjbb2005 on both the original code and our refactored code. We separately run JDBCbench (a test program in HSQLDB) with 100k, 200k, 300k and 400k transactions on both original code and refactored code, and the comparison result is shown in Figure 4. From the chart, we can see that the transaction rate improvement is not obvious when the number of transactions is 100k, and the transaction rate is improved by about 15%, 19%, and 22% when the number of transactions reaches 200k, 300k, and 400k. The results clearly indicate our refactored code has better throughput. Due to space limitation, the results for SPECjbb2005 is not shown. The reason we did not do the experiments for the other three projects is that they do not provide concurrency test cases.

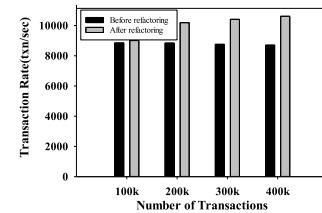


Figure 4: Throughput Comparison

## 5 CONCLUSIONS

This paper presents *FineLock*, an automatic refactoring tool to convert coarse-grained locks into fine-grained locks to reduce lock contention. *FineLock* is evaluated on 5 real-world applications. A total of 1,546 synchronized locks are refactored. Each project takes an average of 27.6 seconds. Experimental results show that *FineLock* can effectively refactor locks.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported, in part by the SRF-Hebei ZD2019093, NSF-Hebei 18960106D and NSF-China 61802166.

## REFERENCES

- [1] Apache. 2019. *Cassandra*. <https://cassandra.apache.org/>
- [2] Bela Ban. 2019. *JGroups*. <http://www.jgroups.org/>
- [3] Standard Performance Evaluation Corporation. 2013. *SPECjbb2005*. <https://www.spec.org/jbb2005/>
- [4] Eclipse. 2020. *Eclipse Java development tools*. <https://www.eclipse.org/jdt/>
- [5] Hypersonic SQL Group. 2019. *HSQLDB - 100% Java Database*. <http://hsqldb.org/>
- [6] IBM. 2018. *The t. j. watson libraries for analysis*. [http://wala.sourceforge.net/wiki/index.php/Main\\_Page](http://wala.sourceforge.net/wiki/index.php/Main_Page)
- [7] Kohsuke Kawaguchi. 2019. *Jenkins*. <https://jenkins.io/>
- [8] B. McCloskey, F. Zhou, D. Gay, and E. A. Brewer. 2006. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, 346–358.
- [9] Benjamin Muskalla. 2008. *Concurrency-related refactorings for JDT*. [https://wiki.eclipse.org/Concurrency-related\\_refactorings\\_for\\_JDT](https://wiki.eclipse.org/Concurrency-related_refactorings_for_JDT)
- [10] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. 2011. Refactoring Java programs for flexible locking. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 71–80.
- [11] Sixth and Red River Software. 2007. *Locksmith*. <https://plugins.jetbrains.com/plugin/1358-locksmith>
- [12] Y. Zhang, S. Dong, X. Zhang, H. Liu, and D. Zhang. 2019. Automated Refactoring for Stampedlock. *IEEE Access* 7, 1 (2019), 104900–104911.

# CPSDebug: A Tool for Explanation of Failures in Cyber-Physical Systems

Ezio Bartocci

Vienna University of Technology  
Vienna, Austria

Niveditha Manjunath

AIT Austrian Institute of Technology  
Vienna University of Technology  
Vienna, Austria

Leonardo Mariani

University of Milano-Bicocca  
Milan, Italy

Cristinel Mateis

AIT Austrian Institute of Technology  
Vienna, Austria

Dejan Nićković

AIT Austrian Institute of Technology  
Vienna, Austria

Fabrizio Pastore

University of Luxembourg  
Luxembourg City, Luxembourg

## ABSTRACT

Debugging Cyber-Physical System models is often challenging, as it requires identifying a potentially long, complex and heterogenous combination of events that resulted in a violation of the expected behavior of the system. In this paper we present CPSDebug, a tool for supporting designers in the debugging of failures in MATLAB Simulink/Stateflow models. CPSDebug implements a gray-box approach that combines testing, specification mining, and failure analysis to identify the causes of failures and explain their propagation in time and space. The evaluation of the tool, based on multiple usage scenarios and faults and direct feedback from engineers, shows that CPSDebug can effectively aid engineers during debugging tasks.

## CCS CONCEPTS

- Software and its engineering → Software verification and validation.

## KEYWORDS

Cyber-Physical Systems, Model-based Development, Testing, Specification Mining, Debugging, Failure Explanation

### ACM Reference Format:

Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, Dejan Nićković, and Fabrizio Pastore. 2020. CPSDebug: A Tool for Explanation of Failures in Cyber-Physical Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404369>

## 1 INTRODUCTION

Cyber-Physical System (CPS) models are complex and heterogenous models that combine discrete and continuous dynamics to precisely capture the behavior of CPSs. Faults in the models must be timely

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3404369>

revealed and fixed, otherwise the successful realization of the CPS can be compromised.

Indeed, debugging can be an extremely challenging task in this domain. Failures might be caused by multiple heterogeneous sources in the model, such as continuous dynamics, switching mechanisms implemented by Finite-State Machine (FSM) components, violated time constraints, incorrect entries in look-up tables and unexpected component interactions. Failures may also propagate in time (i.e., a misbehaviour might be the cause of a later misbehaviour) and space (i.e., a misbehaviour in a component might be the cause of misbehaviours in other components) before they become observable.

Debugging tasks can be supported by *fault-localization* and *failure explanation* techniques. Fault-localization consists of suggesting the possible fault locations based on the program elements executed by passing and failing executions [19]. A popular example is *spectrum-based fault-localization* [1] (SBFL), which has been recently extended to Simulink/Stateflow CPS models [3, 6, 10–12]. Although sometime useful, SBFL techniques have a limited explanatory capability [16], since they provide little overview and explanation of the failure, not really helping engineers assess whether the identified components are faulty and how the failure propagated across the components resulted on an actual failure.

*Failure explanation* techniques support debugging by identifying the sequence of misbehaviors that may explain a failure [2, 5, 14, 17]. These works are not directly applicable to CPS models since they usually leverage the discrete nature of component-based and object-oriented applications that is radically different from the data-flow oriented nature of CPS models, which include analog-mixed signals, hybrid (continuous and discrete) components, and a complex dynamics.

To address this problem, we defined CPSDebug [4], a technique that combines testing, specification mining, and failure analysis to identify the causes of failures in CPS models. CPSDebug outputs a sequence of snapshots that reconstructs how misbehaviors propagate in space and time in the context of the failure. This information can be exploited by engineers to understand the reason of the failure and fix the fault.

This paper describes the tool that implements CPSDebug. It is implemented in MATLAB, it is fully automatic and freely available at <https://gitlab.com/nmanjunath/cpsdebug.git> for download. Early results with an Automatic Transmission Control System and an

Aircraft Elevator Control System, including feedback from engineers who inspected the output of the tool, confirm the value of the tool.

The rest of the paper is organized as follows. Section 2 summarizes how CPSDebug works, and describes the architecture and usage of the tool. Section 3 describes usage scenarios and summarizes the empirical results that we obtained using the tool. We conclude in Section 4.

## 2 CPSDEBUG

This section describes CPSDebug, its architecture and usage.

### 2.1 The Approach

CPSDebug produces failure explanations by *Testing* the CPS model, by *Mining* properties from the data collected during testing, and by *Explaining* the failures based on the violations of mined properties and their relationships. The CPS models input to CPSDebug are MATLAB Simulink/Stateflow models.

The *testing* phase first instruments the software so that the values assigned to every signal and variable of the Simulink model at every timestamp are logged, and then runs the available test cases. A Signal-Temporal Logic (STL) specification [13] is used to distinguish failing and passing test cases, and to classify traces accordingly.

The *mining* phase infers properties from the traces recorded during the execution of passing test cases, to capture how the individual signals and components of the Simulink model behave when no failure is reported. CPSDebug performs two pre-processing steps before mining properties: It identifies correlated signals to drop signals that only provide redundant information; and it groups traces per component (i.e., Simulink block) so that the inferred properties refer to the individual components of the model. These steps are useful to generate properties whose violation can be easily interpreted, because each violation refers to a well defined component. In addition, they avoid the combinatorial explosion of the size of input to the mining tools, which are launched multiple times on small groups of variables rather than on all variables at once. CPSDebug infers two classes of properties from traces: properties that represent the *values that can be assigned to signals* inferred with Daikon [7] and timed automata that capture the *timed state-based behavior of stateful components* inferred with TkT [18]. Failures that can be explained in terms of violations of these properties can be addressed with CPSDebug.

The *explaining* phase exploits the mined properties to analyze the traces recorded during the execution of the failing test cases and generate failure explanations. In particular, CPSDebug collects the signals and variables that violate the mined properties at any point in time and reconstructs the history of the failing execution focusing on the propagation of the misbehaviors. To do this, CPSDebug clusters misbehaviors, to compactly report groups of violations (we refer to these groups of violations as a snapshot), and exploits their time and space distribution, to order in time and localize on the components of the Simulink model the snapshots. The engineer can inspect the snapshots from the first one, usually indicating the faulty component, to the last one to reconstruct the behavior of the software during the failure.

### 2.2 Tool Architecture

Figure 1 illustrates the structure of the tool, which consists of the components described below.

The *Model Instrumenter* is responsible of instrumenting the CPS model under analysis. The objective of instrumentation is to log values assigned to every internal signal and variable in the model. Our Model Instrumenter is implemented as a MATLAB component that inductively navigates the hierarchical structure of the Simulink/Stateflow model in a bottom-up fashion adding tracing capabilities to every element. For every Real, Integer, Boolean value and Enumerated-type signal, it automatically assigns a unique identifier and enables logging of the signal. For every state machine and look-up-table, it adds the logic necessary to record every state transition and access to the table, respectively.

The *Tester* component is a regular off-the-shelves test executor used to run the available test cases. CPSDebug currently uses the MATLAB Simulation Engine to run the test cases. CPSDebug uses an STL formula that is automatically evaluated by a *monitor* against the output traces of the tests. These traces are finally labeled with a *pass* verdict, if the test satisfies the STL formula, or with a *fail* verdict otherwise. CPSDebug uses the RTAMT library [15] and its input language to encode and monitor STL specifications and label the tests with pass/fail verdicts.

The *Property Miner* selects the traces labeled with a *pass* verdict and uses them for mining properties. It is also responsible of dropping the correlated signals and grouping traces per component (i.e., per Simulink/Stateflow block). Our tool is designed to integrate and run a number of mining solutions. The current version integrates Daikon [7] and TkT [18] to derive universal and real-time state-based properties.

The *Monitor + Trace Diagnoser* component produces failure explanations by analyzing the collected failing traces. In this step, when the trace is evaluated against the mined properties, a list of signals that violate the properties and the time at which the signals first violate the properties are reported. CPSDebug uses RTAMT to detect violations of universal properties and TkT to detect violations of real-time state-based properties.

The *Failure Mapper* clusters the fail-annotated signals by their violation times and maps them to their corresponding model blocks, i.e., to the model blocks that the fail-annotated signals originate from. The violated signals with their corresponding origin blocks are used to create a sequence of snapshots, one for each cluster of property violations. The designer can inspect the sequence of snapshots to reconstruct the anomalous behaviors that have been observed during the failing execution, and identify both the root events and root components responsible for the investigated problem.

### 2.3 Tool Usage

CPSDebug can be executed in any environment equipped with Java 8, Python and MATLAB 2016b to 2018b. It fully supports CPS models designed in MATLAB Simulink/Stateflow, but it does not support models exploiting other MATLAB toolboxes, such as Simscape/SimEvents. The tool can be executed from the command line to be easily integrated within automated scripts.

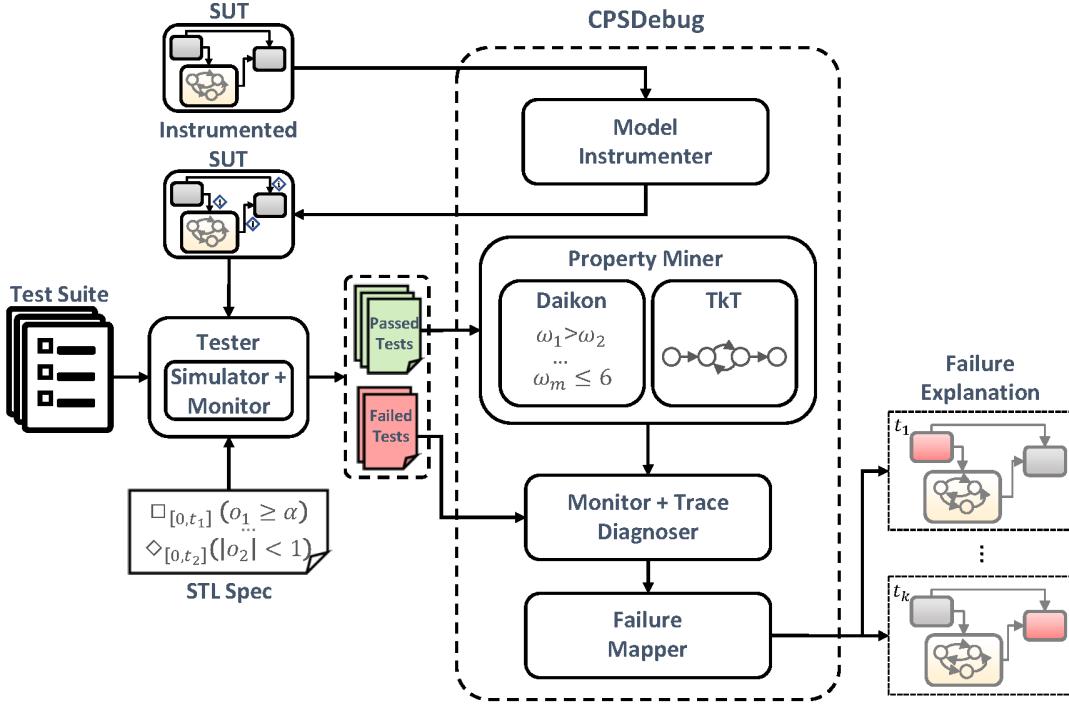


Figure 1: Structure of CPSDebug

The behavior of the tool is influenced by a configuration file that can be used to specify parameters. To launch the tool, it is necessary to specify the model under test, the STL specification used as test oracle and the test suite.

The tool produces as failure explanation output a table for each failed tests. The table stores in tabular form an ordered sequence of the identified snapshots. The data for each snapshot include the signals that violated the mined properties, the time of violation, the block that the signal belongs to and the violated properties. In addition, the tool provides statistics about the executed tests and the mined properties, such as the number of passed and failed test, and the number of properties inferred with Daikon and TkT.

### 3 TOOL ASSESSMENT

We experimented CPSDebug in the context of both coverage-based and regression testing [4]. We experimented the former scenario with the Automatic Transmission Control System (ATCS) [9] and the latter scenario with the Aircraft Elevator Control System (AECS) [8] model. We ran the experiments on Intel i7-8650 2.11GHz with 8 cores and 32GB RAM. The models chosen to assess CPSDebug contain both continuous and discrete blocks with hierarchies. They are publicly available models of medium complexity that are representative of industrial models with regard to behavioural dynamics.

*Coverage-based testing.* ATCS is a model of an automatic transmission controller that exhibits both continuous and discrete behavior. The system has two inputs, the throttle  $u_t$  and the break  $u_b$ , and two continuous-time state variables, the speed of the engine  $\omega$  (rpm), the speed of the vehicle  $v$  (mph) and the active gear  $g_i$ . When the

system is started, both the vehicle speed and the engine speed are initialized to 0. It follows that the output trajectories depend only on the input signals  $u_t$  and  $u_b$ , which can take any value between 0 and 100 at any point in time.

The overall ATCS model consists of 55 signals, 4 look-up tables and 2 FSMs running in parallel with 3 and 4 states, respectively. The available coverage-based test suite consists of 100 tests implemented to cover the functionalities of the system. We experimented CPSDebug with two injected faults: (1) a faulty transition in a FSM, and (2) an altered entry in a look-up table.

The outcome of CPSDebug is the set of three signals that explain the failure, two concerning the *throttle* and one concerning the *engine torque*, all having values higher than expected. The signal about the *engine torque* is the output of the component in which the fault was injected, while the two signals about the throttle are the inputs that trigger this anomalous behavior. Since both the triggers of the anomalous behavior and the anomalous output generated by the faulty component have been identified, it is straightforward for the engineer to identify the right component to inspect and reveal the fault. Indeed, failure explanation reduces the scope of the analysis by 95% allowing the engineers to focus on a small subset of the suspicious signals.

*Regression testing.* The architecture of an AECS contains one elevator on the left and one on the right side, with redundancy. Each elevator is equipped with two hydraulic actuators. Both actuators can position the elevator, but only one shall be active at any point in time. There are three different hydraulic systems that drive the four actuators. The system uses state machines to coordinate

the redundancy and assure its continual fail-operational activity. This model has one input variable, the input Pilot Command, and two output variables, the position of left and right actuators, as measured by the sensors.

The AECS model consists of 426 signals, from which 361 are internal variables that are instrumented (279 real-valued, 62 Boolean and 20 enumerated - state machine - variables) and any of them, or even a combination of them, might be responsible for an observed failure. We experimented AECS with a fault each in *hydraulic system2* and *hydraulic system3* components. These example faulty components are available with AECS.

The fault in the *hydraulic system 3* in AECS was injected at time 2 and the fault in *hydraulic system 2* was injected at time 5. The output of CPSDebug is a set of 51 signals partitioned into two snapshots. The first snapshot consists of 50 signals that show an anomaly at time 2 and the second snapshot consists of 1 signal that shows an anomaly at time 5. The two snapshots contain output signals from the faulty hydraulic systems 3 and 2, respectively. The engineer simply focuses on the components responsible of the anomalous signals in the two snapshots. It follows that the failure explanation reduces the scope of the investigation performed by the engineer by 88% while detecting the root cause of the failure, thus helping the engineer to quickly identify faulty components.

Out of 171 tests, 166 tests were labeled as fail after the faulty components were introduced in the system. This large number of failures is due to a rigid specification that has strong assumptions about the environment. This setting has been useful to experiment the usage of the tool when most of the tests fail and little information is available to generate properties.

*Feedback from Professionals.* To confirm that the output produced by the tool is indeed useful to engineers, we submitted the output of the tool to 3 engineers from major tool vendors and automotive OEMs asking them to evaluate the outcomes of our tool for a selection of faults. The engineers found the tool potentially useful since debugging of CPS models can be very difficult. They shared appreciation for the generated output and they were willing to experiment the tool with their projects. More details about our experimental evaluation can be found in the paper by Bartocci et al. [4].

## 4 CONCLUSION

Debugging faults in Cyber-Physical System models can be extremely challenging due to the complex interactions between discrete and continuous behaviors. Indeed, an anomalous behavior may propagate through many components before resulting in an observable failure. Reconstructing the complex and potentially long chain of events responsible of a failure is of fundamental importance to localize, understand and fix the fault.

To address this challenge, we presented CPSDebug, a tool for explaining failures in MATLAB/Simulink models. The tool combines testing, specification mining, runtime verification and clustering techniques to derive meaningful explanations, which consist of sequences of snapshots that reconstruct the propagation in time and space of the anomalous events responsible for the failure, offering a clear starting point for the analysis of the failure and the correction of the fault.

As part of our future work, we plan to improve CPSDebug along two lines. We will first enhance the tool usability by directly integrating it in MATLAB, so that it can be launched and used without leaving the programming environment. Moreover, we plan to investigate the possibility to integrate guided test generation strategies designed to improve the explanations.

## REFERENCES

- [1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques*. IEEE, 89–98.
- [2] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. 2009. AVA: Automated Interpretation of Dynamically Detected Anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ACM, 237–248.
- [3] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Nickovic. 2018. Localizing Faults in Simulink/Stateflow Models with STL. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control*. ACM, 197–206.
- [4] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Nickovic. 2019. Automatic Failure Explanation in CPS Models. In *Software Engineering and Formal Methods (LNCS)*, Vol. 11724. Springer, 69–86.
- [5] Mitra T. Befrouei, Chao Wang, and Georg Weissenbacher. 2016. Abstraction and Mining of Traces to Explain Concurrency Bugs. *Formal Methods in System Design* 49, 1–2 (2016), 1–32.
- [6] Jyotirmoy V. Deshmukh, Xiaoqing Jin, Rupak Majumdar, and Vinayak S. Prabhu. 2018. Parameter optimization in control software using statistical fault localization techniques. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. IEEE, 220–231.
- [7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45.
- [8] Jason Ghidella and Pieter Mosterman. 2005. Requirements-based testing in aircraft control design. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*.
- [9] Bardh Hoxha, Houssam Abbas, and Georgios Fainekos. 2015. Benchmarks for Temporal Logic Requirements for Automotive Systems. In *ARCH14-15. 1st and 2nd International Workshop on Applied verIification for Continuous and Hybrid Systems (EPiC Series in Computing)*, Vol. 34. 25–30.
- [10] Bing Liu, Lucia Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 359–370.
- [11] Bing Liu, Lucia Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Localizing Multiple Faults in Simulink Models. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 146–156.
- [12] Bing Liu, Lucia Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 (2016), 431–459.
- [13] Oded Maler and Dejan Nickovic. 2013. Monitoring properties of analog and mixed-signal circuits. *Software Tools for Technology Transfer* 15, 3 (2013), 247–268.
- [14] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering (TSE)* 37, 4 (2011), 486–508.
- [15] Dejan Nickovic and Tomoya Yamaguchi. 2020. RTAMT: Online Robustness Monitors from STL. arXiv:2005.11827 [cs.LO]
- [16] Chris Parnin and Alex Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 199–209.
- [17] Fabrizio Pastore, Leonardo Mariani, Antti E. Johannes Hyvärinen, Grigory Feduyukovich, Natasha Sharygina, Stephan Schestedt, and Ali Muhammad. 2014. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 37–48.
- [18] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. 2017. Timed k-Tail: Automatic Inference of Timed Automata. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 401–411.
- [19] Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

# Test Recommendation System Based on Slicing Coverage Filtering

Ruixiang Qian

State Key Laboratory for Novel  
Software Technology, Nanjing  
University, Nanjing, China

Yang Feng

State Key Laboratory for Novel  
Software Technology, Nanjing  
University, Nanjing, China

Yuan Zhao

State Key Laboratory for Novel  
Software Technology, Nanjing  
University, Nanjing, China

Duo Men

State Key Laboratory for Novel  
Software Technology, Nanjing  
University, Nanjing, China

Qingkai Shi

Hong Kong University of Science and  
Technology, Hong Kong, China  
qingkaishi@gmail.com

Yong Huang

Zhenyu Chen  
Mooctest Inc., Nanjing, China  
chenzhenyu@mooctest.com

## ABSTRACT

Software testing plays a crucial role in software lifecycle. As a basic approach of software testing, unit testing is one of the necessary skills for software practitioners. Since testers are required to understand the inner code of the software under test(SUT) while writing a test case, testers usually need to learn how to detect the bug within SUT effectively. When novice programmers started to learn writing unit tests, they will generally watch a video lesson or reading unit tests written by others. These learning approaches are either time-consuming or too hard for a novice. To solve these problems, we developed a system, named TeSRS, to assist novice programmers to learn unit testing. TeSRS is a test recommendation system which can effectively assist test novice in learning unit testing. Utilizing program slice technique, TeSRS has gotten an enormous amount of test snippets from superior crowdsourcing test scripts. Depending on these test snippets, TeSRS provides novices a easier way for unit test learning. To sum up, TeSRS can help test novices (1) obtain high level design ideas of unit test case and (2) improve capabilities(e.g. branch coverage rate and mutation coverage rate) of their test scripts. TeSRS has built a scalable corpus composed of over 8000 test snippets from more than 25 test problems. Its stable performance shows effectiveness in unit test learning.

Demo video can be found at <https://youtu.be/xvrLdvU8zFA>

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging.

## KEYWORDS

Program Slice, Static analysis, Test recommendation, Test guidance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07.

<https://doi.org/10.1145/3395363.3404370>

## ACM Reference Format:

Ruixiang Qian, Yuan Zhao, Duo Men, Yang Feng, Qingkai Shi, Yong Huang, and Zhenyu Chen. 2020. Test Recommendation System Based on Slicing Coverage Filtering. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3395363.3404370>

## 1 INTRODUCTION

With the development of software engineering in the past few decades, software has become more and more intelligent. With the architecture has become more complex, software is becoming more error-prone at the same time. Unit testing is an important white box test approach, which is almost a compulsory course to every software practitioners. When doing unit testing, testers are required to understand the inner code of the software under test(SUT) entirely in order to write unit test cases(test scripts) to detect latent bugs within the software. However, it is not easy for a novice to write test scripts in an effective and elegant way. It usually takes them a long-term to practise and learn to obtain the high level design idea of unit test cases. Therefore, how to efficiently guide a novice is of great significance.

According to the *State of Testing Survey 2019*<sup>1</sup> initiated by Practitest with about 1,000 participants from more than 80 countries, 65% participants learn testing by "just doing it", and more than 40% participants learn testing through formal training or certifications and courses. This survey implies that practice is a main approach of learning testing and testers are usually longing to improving test skills through formal software testing courses. However, these approaches are either time-consuming or too hard for a novice.

In addition to courses and practice, testers also utilize automated test generation tools(e.g. Evosuite<sup>[4]</sup>) as well as visualized coverage tools (e.g. Jacoco<sup>[5]</sup>) to assist testing. Automated test generation tools can generate test suites automatically with few or without human intervention. Coverage tools (e.g. Jacoco) can detect the coverage rate (e.g. branch coverage) of test suites, revealing the unreach parts of an SUT to testers in order to guide them refine test scripts. Although these tools can effectively help experienced testers improve test quality and reduce their workload, these tools are usually not suitable for novices due to lack of experience.

<sup>1</sup>[https://qablog.practitest.com/wp-content/uploads/2019/06/STOT\\_2019\\_ver1\\_2.pdf](https://qablog.practitest.com/wp-content/uploads/2019/06/STOT_2019_ver1_2.pdf)

Therefore, we design and implement a system named TeSRS to save novice testers from the above situations. TeSRS is test recommendation system which is now deployed on MoocTest<sup>2</sup> platform. MoocTest has been devoted to education of software testing in the past few years. Through holding national software test competition<sup>3</sup>, MoocTest has collected an enormous amount of test scripts. In order to get a higher score, participants usually need to modify their test scripts several times before final submission, which ensures the quality of these test scripts. Using these test scripts as raw material, we adopt static program slice[1, 10] technique to build a test snippets corpus. Since these test scripts are written for competition, some of them may more or less exist coupling. We slice coupled test scripts into fine-grained test snippets which are more suitable for recommendation. TeSRS associates each test snippet with test criteria (e.g. branch coverage) and provides online recommendation service. When test learners encounter a bottleneck while writing a test script, TeSRS will recommend associated test snippets to them. The recommended test snippets can help test learners improve test quality notably, which will eventually reflect in their score. Since test snippets are recommended while test learners are coding, it is much easier for them to understand the test logic behind these snippets. In this paper, we mainly make the following contributions:

- Building a test snippets corpus by slicing test scripts supplied by MoocTest, which consists of over 8000 test snippets from 25 testing problems.
- Proposing a test recommendation system, named TeSRS, to help test novices learn testing efficiently and effectively through interactive online test snippet recommendation.

## 2 TESRS

TeSRS is a test recommendation system now is embedded in the MoocTest WebIDE, which can recommend fine-grained and easy-to-understand test snippets to test learners online. The overview process of TeSRS is shown in Fig.1.

TeSRS is composed of two parts: (1) Test snippets corpus. We build the corpus in two phases: 1) Slice phase. Utilizing off-the-shelf program slice technique, we split the test scripts supplied by MoocTest into fine-grained and easy-to-understand test snippets. 2) Association phase. To associate each test snippet with test criteria, we compute test capability for each test snippet, and build the mapping relation between snippet and corresponding test criteria. (2) Interactive test recommendation component. When test learners encounter a bottleneck while writing test scripts, they can ask TeSRS for help. TeSRS will recommend top K test snippets to test learners according to the dissimilarity between test snippets and incomplete scripts. We will talk about these two parts elaborately in the following two subsections.

### 2.1 Test Snippet Corpus

Test snippet corpus is the basis of TeSRS. We have built a scalable corpus containing 8000 test snippets from 25 test problems. The building process can be divided into the following two phases:

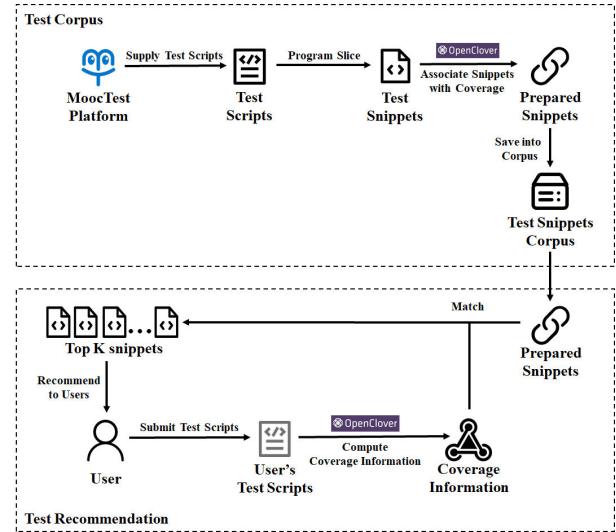


Figure 1: Overview of TeSRS



Figure 2: Test script to test snippets

**Slice phase** MoocTest has collected an enormous amount of superior test scripts(i.e. test scripts which got a high score in competition). All of these test scripts are Junit<sup>4</sup> tests from standard maven<sup>5</sup> projects. A example test script is shown at the top of fig.2. Although a test script may perform well in competition, it is usually coupled and hard to read, which are not suitable for test recommendation. Therefore, we use slice algorithm provided by Wala[1] to decouple test scripts and get more fine-grained test snippets. Wala is a static program analysis library for Java and Javascript, which also provide off-the-shelf backward and forward slice methods. Wala requires a statement, which is called "seed" in wala context, as the slice point to drive a slice process. Standard Junit test methods generally use static assert methods, such as *Assert.assertEquals*,

<sup>4</sup><https://junit.org>

<sup>5</sup><https://maven.apache.org/>

to validate the correctness of outputs. An invocation of an assert method usually implies a completion of a testing phase. Therefore, we treat each assert method as a "seed" and slice test scripts in a backward style. The test snippets are shown at the bottom of fig.2.

**Association phase** This phase aims to bind each test snippet with some certain test criteria. In this paper, we bind each test snippet with branch coverage rate by OpenClover<sup>6</sup>. OpenClover is a coverage report generation tool which can generate coverage report in xml format at runtime. Association is completed in four steps: (1) Construct blank test patterns. Since a test snippet are extracted from a test script, it must have the same dependency (e.g. outer classes which should be imported) as the corresponding script. For each test snippet, we firstly find its original project, and then locate the class where it comes from. After that, we construct a blank test pattern by deleting all the test methods within this class. Finally, we replace the pom.xml of this project by a pom.xml contains dependency for OpenClover. (2) Insert snippets into patterns. We firstly complete the method signature of each test snippet by JavaParser[6], and then insert it into blank test pattern accordingly. Through insertion, we get executable temporary test projects(test temp for short) and ready for collection. (3) Collect coverage information. We execute the test temps we get in (2) and run OpenClover to collect coverage report. (4) Associate test snippet with coverage rate. Since xml file is usually inconvenient for persistence and not suitable for recommendation, we transform coverage reports we get in (3) into JSON format, and then associate each test snippet with its JSON coverage information accordingly. These test composites will be finally saved into database.

## 2.2 Test Recommendation Component

Test recommendation component provide online recommendation service, the work flow is shown in fig.3. When test learners submit their test snippets on WebIDE, test recommendation component will collect coverage information from submitted test snippets and recommend test snippets to them timely. Test Snippets recommendation process consists of the following two phases:

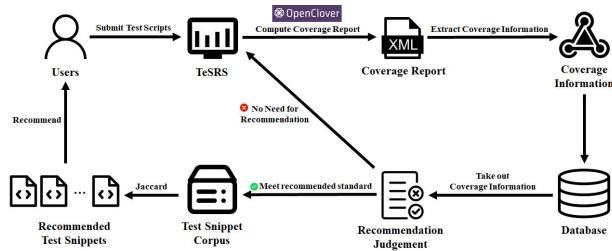


Figure 3: Recommendation Work Flow

**Collection Phase** When users submit and run their test scripts online, recommendation component will firstly utilize OpenClover to generate coverage report, which records coverage rate for each method under test(MUT). Then these coverage rates will be extracted respectively and stored into database according to MUTs.

**Recommendation Phase** Test recommendation component won't recommend test snippets to users at the very beginning.

<sup>6</sup><https://openclover.org>

Recommendation service will be triggered when 1) the user has successively submitted for several times 2) and the number of submissions has exceeded our predefined threshold  $N$ (which is one in this paper) and 3) the quality of test scripts, which is reflected by test score, has no improvement. Recommendation service will be completed in three steps. (1) Firstly, recommendation component finds all related test snippets according to the signature of each MUT. (2) Secondly, recommendation component builds test vectors for each test script which is submitted by user and test snippet which is taken out from corpus. A test vector  $v$  is a representation of coverage rate and each  $v$  correspond to a MUT. At present, we only adopt branch coverage as the accordance for recommendation. Therefore, the test vector of a MUT  $m$  is defined as  $v_m = [b_1, b_2, b_3, \dots, b_n]$  where  $n$  denotes the number of branches and  $b_i (1 \leq i \leq n)$  denotes the evaluation result of a branch of  $m$ . The value of  $b_i (1 \leq i \leq n)$  is *true* or *false*, representing whether the  $i$ th branch of  $m$  has been covered or not. We use  $v_{msc}$  and  $v_{msn}$  to represent test vector of a script and a snippet respectively. (3) Finally, we compute Jaccard distance between  $v_{msc}$  and  $v_{msn}$  and recommend top  $k$  test snippets according to it. The math formula of Jaccard distance is shown below:

$$d_j = 1 - \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

$A$  and  $B$  are both sample sets. Jaccard distance measures the dissimilarity between two sets. We treat each test vector as a set, then the Jaccard distance of  $v_{msc}$  and  $v_{msn}$  is:

$$d_j = 1 - \frac{|v_{msc} \cap v_{msn}|}{|v_{msc} \cup v_{msn}|} = 1 - \frac{l}{n}$$

Note that  $l$  denotes the number of branches shared by  $v_{msc}$  and  $v_{msn}$ , then  $d_j$  can be simplified to the ratio of the number of different branches and the number of total branches. Recommendation component will recommend top  $K$  test snippets to user to help them to improve test score. We set  $K$  to 2 at present. If there are more than 2 test snippets got the highest  $d_j$ , which means they are worthy recommending equally, component will randomly select 2 test snippets to recommend. The insight behind this strategy is that each of these test snippet is able to improve test score by covering new branches, so it doesn't matter which 2 will be recommended to users.

## 3 EXPERIMENT

### 3.1 Research Questions

To evaluate the effectiveness of TeSRS, we summarize key problems into the following research questions and design experiment to answer them respectively:

- **RQ1:** Can test snippets maintain the test quality of their origin test scripts?
- **RQ2:** Can TeSRS help test novice with test learning effectively?

### 3.2 The Reliability of Test Snippets

We select four test problems(AStar, ALU, Triangle and RBTree) to answer RQ1. To verify whether test snippets maintain the high quality of their original test scripts, we measure the test criteria(i.e. branch coverage and mutation coverage) of each test script(as

shown in "origin" column) and test snippet(as shown in "slice" column) respectively. We make test snippets executable through the following operations: (1) Wrap test snippets into independent test methods. (2) Add outer dependencies (i.e. import statements like `import java.lang.*`) and (3) add inner dependencies (e.g. inner classes and static methods). These operations are mainly done by a simple program. However, loss of several lines of code will inevitably occur when program is relatively complicate(e.g. has exception handler statements). We mended these loss manually. The experiment result is shown in table.<sup>1</sup> We can find test quality doesn't change in terms

**Table 1: Test criteria before and after slicing**

Problem	Origin		Slice	
	Branch	Mutation	Branch	Mutation
Astar	81.0%	75%	81.0%	72%
ALU	81.7%	79%	81.7%	79%
RBTREE	92.0%	83%	92.0%	82%
Triangle	81.2%	78%	81.2%	78%

of branch coverage, while tiny decrease in mutation coverage can be observed on some test problems(i.e **Astar** and **RBTREE**). This is because program slice may destroyed some test logic while decoupling original test scripts. Some latent bugs need specific test sequences to trigger. However, since the decrease in percentage is not that huge and users will reunion these test snippets while they are coding with help of TeSRS, we will create similar test sequence so that the tiny loss of mutation coverage rate won't affect the quality of recommended test snippets.

### 3.3 The Effectiveness of TeSRS

We ask 20 students to respectively finish one test problem with the help of TeSRS and collect their feedback in the form of questionnaire. We mainly investigated the following aspects: (1) Test script and test snippet, which one is more readable, and (2) user experience. In terms of the first aspect, we give out 3 test scripts and 3 test snippets. For each script or snippet, participant should give out a score about its readability from 1 to 5. Higher score implies a higher readability. As a result, test scripts got **180** in total while test snippets got **253**. This indicates program slicing does improve the readability of recommendation content by filtering out less important code. In terms of the second aspect, we ask every participants to evaluate TeSRS in two aspects: (1) To what extent can TeSRS widen their test idea? (2) To what extent can TeSRS help them with unit testing learning. Participants also need to give out a score in 1 to 5. As a result, all participants give out a score larger than 3 for both of these two aspects. Moreover, half of the participants give 5 point when evaluate the first aspect, 12 participants give a 5 at the second evaluation. Both of these two evaluation result indicates the effectiveness of our system.

## 4 RELATED WORK

Code recommendation can help developers find desired code from thousands of software artifacts and improve users' productivity. This problem is generally treated as an context-based query problem in information retrieval(Antunes et al. [3]). Rahman and Roy[9] exploit code examples collected from *Github* to help developers

build a software with robust exception handle mechanism. In 2019, Ai et al.[2] proposed a code statement sequence information based recommendation system, named SENSORY, to help developers to implement unfamiliar programming tasks. Instead of directing developers to write a test script properly, all of the above techniques focus on assisting developers to use APIs better.

In terms of test code recommendation, Janjic and Atkinson.[7] leverage lessons learned from traditional software reuse to proactively make test recommendation. The core of their recommendation system is SENTRE(Search-ENhanced Testing with REuse), which can make reverse search for Junit test cases. Pham et al.[8] assist test novices in test learning by strategically showing them contextual test code examples from a project's test suite. Although these two techniques share similar insight with our approach, they did not utilize program slice technique to refine the code for recommendation or provide test learners a online platform for practice.

## 5 CONCLUSION

In this paper, we propose a test recommendation system, named TeSRS, to help test novices learn unit tests effectively and efficiently. We built a test snippet corpus, which comprises over 8000 test snippets from more than 25 test problem, by slicing the test scripts supplied by MoocTest. We also implemented a test recommendation component which can interactively recommend suitable test snippets to users. The reliability of test snippets and the effectiveness of recommendation component has been demonstrated through experiments. In the future, we plan to enlarge our test snippet corpus and further complete recommendation mechanism.

## ACKNOWLEDGEMENTS

This work is partially supported by the National Natural Science Foundation of China(61802171) and the Key Project of Natural Science Research in Anhui Higher Education Institutions(KJ2019ZD67).

## REFERENCES

- [1] 2019. *T.J. Watson Libraries for Analysis*. <https://github.com/wala/WALA/wiki>
- [2] Lei Ai, Zhiqiu Huang, Weiwei Li, Yu Zhou, and Yaoshen Yu. 2019. SENSORY: Leveraging Code Statement Sequence Information for Code Snippets Recommendation. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 27–36.
- [3] Bruno Antunes, Barbara Furtado, and Paulo Gomes. 2014. Context-based search, recommendation and browsing in software development. In *Context in Computing*. Springer, 45–62.
- [4] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
- [5] Marc R Hoffmann, B Janiczak, and E Mandrikov. 2011. EclEmma-jacoco java code coverage library.
- [6] Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.
- [7] Werner Janjic and Colin Atkinson. 2013. Utilizing software reuse experience for automated test recommendation.
- [8] Raphael Pham, Yauheni Stolar, and Kurt Schneider. 2015. Automatically recommending test code examples to inexperienced developers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 890–893.
- [9] Mohammad Masudur Rahman and Chanchal K Roy. 2014. On the use of context in recommending exception handling code examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 285–294.
- [10] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

# Automated Mobile Apps Testing from Visual Perspective

Feng Xue

School of Computer Science and Engineering

Northwestern Polytechnical University

Xi'an, PR China, 710072

xue.f@mail.nwpu.edu.cn

## ABSTRACT

The current implementation of automated mobile apps testing generally relies on internal program information, such as reading code or GUI layout files, capturing event streams. This paper proposes an approach of automated mobile apps testing from a completely visual perspective. It uses computer vision technology to enable computer to judge the internal functions from the external GUI information of mobile apps as we humans do and generates test strategy for execution, which improves the interactivity, flexibility, and authenticity of testing. We believe that this vision-based testing approach will further help alleviate the contradiction between the current huge test requirements of mobile apps and the relatively lack of testers.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging

## KEYWORDS

Software testing, Test automation, Mobile applications, Computer vision

### ACM Reference format:

Feng Xue. 2020. Automated Mobile Apps Testing from Visual Perspective. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20), July 18-22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 5 pages.  
<https://doi.org/10.1145/3395363.3402644>

## 1 INTRODUCTION

In recent years, mobile apps have shown a rapid development trend due to benefits from the advancement of mobile devices and mobile technology. As of 2019, the total number of mobile apps on Google Play and Apple App Store had reached 4.4 million [1]. Furthermore, from the advent of mobile apps in 2009

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA'20, July 18-22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3402644>

to 2019, the proportion of traffic generated by mobile devices increased from 0.7% to 52.6% of global website traffic [2]. Mobile apps are gradually replacing desktop software in people's daily lives.

However, unlike desktop software, mobile apps are usually large in number, diverse in type, with frequent version iterations, and have more complex operating environment [3, 4]. These features of mobile apps make testing more difficult, and it is even more urgent to seek effective automated testing approaches to solve the current mobile apps testing dilemma. At present, the main automated testing techniques for mobile apps can be divided into the following three levels.

The first level techniques focus on automating test execution.

Script-based testing enables the automatic execution of test cases defined by manually editing scripts. The test automation frameworks, such as Espresso [5], XCTest [6], Appium [7], can convert scripts into event streams and input AUT (App Under Test) to simulate test execution. Different from text-only script, Sikuli [8], Eyeautomate [9] allow to define a script containing visual information (a screenshot of interface element) and implement visual GUI testing through image matching.

Record-replay testing records the manually executed test cases and then plays them back for reuse of test execution. RERAN [10] captures the low-level event stream including GUI events and sensor events, then replays them for test execution. SARA [11] uses the proposed self-replay and adaptive replay mechanisms to achieve a high record-replay success rate for single and cross devices. LIRAT [12] combines image processing technology to achieve cross-platform record-replay testing.

The second level techniques are based on the first level and focus more on automating test case generation and optimization.

Random testing uses a random strategy to explore and test apps. Although random, it automatically generates test inputs. Monkey [13], a native testing tool for Android, sends randomly selected GUI events and system events for testing. More improvements in this approach revolve around increasing the efficiency of effective event generation [14, 15].

Model-based testing automates testing by modeling apps behavior and using model generated test cases. MobiGUITAR [16] dynamically traverses AUT to build model and then generates test cases. AMOGA [17] takes a static-dynamic approach to statically obtain the association between GUI elements and events from the source code, and then dynamically and orderly explores AUT to generate model. These models are usually represented by a finite state machine.

Search-based testing leverages heuristic algorithms to generate test cases and optimize test sequences. AGRippin [18] founded by genetic and hill climbing algorithms facilitates efficient and effective test case generation. Sapienz [19] introduces Pareto multi-objective approach to reach better test coverage with fewer test cases.

Transfer-based testing is dedicated to cross-platform and cross-app use of test cases. TestMig [20] uses similar comparison of GUI controls and events to realize the migration of test cases from IOS to Android. AppTestMigrator [21] and CraftDroid [22] enable test cases to run on multiple apps with similar functions through matching the semantically similar of functions.

The third level techniques further introduce the feature of learning.

Traditional model learning-based testing is based on the traditional finite state machine modeling to add a learning mechanism to achieve dynamic construction and generate on-the-fly test inputs. SwiftHand [23] employs an improved L\* algorithm to refine model during test execution. DroidBot [24] dynamically updates the model through the adapter module and the brain module.

Deep learning-based testing applies deep neural network technology to train an end-to-end test model. Humanoid [25] designs a deep neural network model to generate human-like inputs by learning user interaction with apps. DL-Droid [26] proposed a deep learning system to detect malicious Android apps through dynamic analysis using stateful input generation.

Reinforcement learning-based testing develops a reinforcement learning mechanism for mobile apps to automatically learn to obtain a test model by a trial-and-error way. [27, 28] exploit Q-learning algorithm to explore AUT and build test model.

All these automated testing techniques have significantly promoted the development of mobile apps testing. However, their implementation is more or less dependent on the internal information of apps, which leads to the limitations in use. Thus, we propose a vision-based testing approach that simulates the user's cognitive and testing process of mobile apps from a totally visual perspective, enabling computer to learn and understand functions contained in mobile app interfaces and associate the correct testing actions. Finally, it can test mobile apps visually like humans without being limited by software.

This approach aims to achieve a more abstract cognition of mobile apps by virtue of the advantages of visual technology (different from most testing techniques that explore and learn mobile apps from a software perspective). Then, it uses the functions of apps as object for modeling to train the computer to learn to cognize the internal function under the external appearance of GUIs among multiple apps (different from most testing techniques use AUT as object for modeling). For example, most apps include a log-in function and the log-in operation is almost similar, yet the log-in GUIs have different styles in different apps. Our approach gives the computer the ability to abstract the representative features of the log-in interface and thus help it to determine the log-in function across apps but not just to identify the log-in of a certain AUT. Subsequently, the

computer can associate the appropriate test actions to test certain functions in multiple apps, just as we human testers or users do.

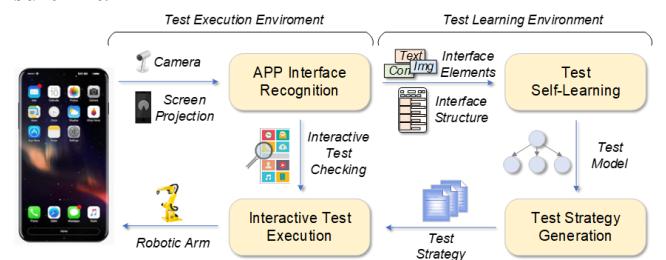
Our expected contributions for this vision-based automated testing approach are followed:

- Recognize apps completely visually like user without resorting to any internal program information.
- Build a more generalized test model with respect to the test model for certain AUT to solve cross-platform and cross-version issues.
- Achieve flexible interactive judgment testing rather than rigid automated execution.

## 2 VISION-BASED AUTOMATED TESTING OVERVIEW

An overview of the vision-based automated testing we propose is presented in this section. Figure 1 shows the main modules of the testing approach and an overview of the testing process. These modules are (1) *App Interface Recognition* and (2) *Interactive Test Execution* belonging to *Test Execution Environment*, and (3) *Test Self-Learning* and (4) *Test Strategy Generation* belonging to *Test Learning Environment*.

*Test Execution Environment* completes the test execution of mobile apps in an automated manner and the collection of visual information of apps. In order to reduce the dependence on platform, here we use robotic arm that mimics realistic human actions on mobile apps [29, 30] for test execution and use camera shooting or screen projection to obtain GUI visual information. *Test Learning Environment* implements learning of mobile apps through the GUI visual information, and training to generate corresponding test strategy. A learning-based test model will be built in it.



**Figure 1: Overview of the vision-based automated testing process**

**App Interface Recognition.** *App Interface Recognition* adopts a purely visual way to recognize apps. Compared with the way of constructing test models by reading configuration files or source code, using vision to cognitive app: on the one hand, from app's external function manifestation rather than internal code logic to explore the app, can achieve a more human-like testing; on the other hand, for the specificity of multi-platform and multi-version of mobile apps, visual way can provide a cross-platform testing advantage and facilitate abstract and flexible judgment. *App Interface Recognition* recognizes the

type and location information of app interface elements and extracts interface features to form GUI abstraction.

**Test Self-Learning.** *Test Self-Learning* further cognizes app structure based on app interface elements recognized by *App Interface Recognition*. According to the characteristics of app and the commonalities of app function, two aspects of abstract cognition are respectively conducted: one is the appearance of single static pages, and the other is the dynamic behaviors between multiple pages. Further let computer learn to judge functions of app from the external information and associate accurate test actions. Then, app function-based models are built.

**Test Strategy Generation.** As the test model generated by *Test Self-Learning* is a function-based model, it cannot be used directly for the AUT. *Test Strategy Generation* transforms function-based models into app type-based models and generates test strategy for AUT.

**Interactive Test Execution.** *Interactive Test Execution* is performed on the test environment throughout the entire testing process. After executing the test actions of each step, the interactive test execution calls the *App Interface Recognition* to obtain the response interface, check the correctness of the test execution and judge whether the interface shows an abnormality by comparing it with test strategy.

### 3 IMPLEMENTATION OF VISION-BASED AUTOMATED TESTING

According to the vision-based automated testing process we designed, there are two key issues to break through. First, how to effectively capture GUI information from a visual perspective and achieve an abstract description of GUI; second, how to possess the generalized cognitive ability of app interface and the association ability of appropriate actions like humans. Figure 2 shows the technical approach for the vision-based automated testing, which contains two steps *3.1 GUI Abstraction* and *3.2 GUI Cognition and Action Association* to solve the two issues.

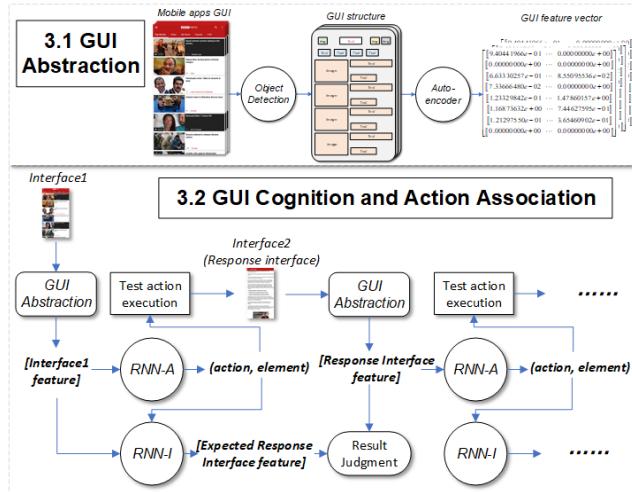


Figure 2: Technical approach for the vision-based automated testing

### 3.1 GUI Abstraction

For the GUI of mobile apps, we do not care about the styles of the GUI, but the GUI controls it contains. Therefore, an object detection technique is applied to conduct interface element recognition and obtain the abstract GUI structure. In view of the dynamic and real-time nature of testing, a one-stage object detection technique would be more appropriate, such as yolov3 [31] we used in the environment, which allows a balance between speed and precision.

We divide common GUI elements into 12 types (TextView, EditText, ImageView, Button, ImageButton, RadioButton, ToggleButton, CheckBox, Spinner, Switch, ProgressBar, NumberPicker) from a visual point of view and build a mobile apps GUI dataset for one-stage object detection based on RICO [32] a large dataset of mobile apps interfaces. In the training of interface element recognition, we improved the loss function and result output judgment of yolov3 to make it more focused on accurate bounding box output, because the omission of element detection is more fatal to the test than misclassification. Consequently, we extract GUI structure only by visual means without resorting to other platform-dependent tools. In the generated GUI structure, different types of GUI elements will be distinguished by colors. Using GUI structure instead of the original interface image will help the computer to better determine the meaning of the interface elements in app interface and can lead the robotic arm to perform accurate actions on interface.

After obtaining the GUI structure, we expect that the computer can further extract the image features of GUI structure. Here, an unsupervised autoencoder neural network [33] is used to obtain the feature vectors of GUI structure. Thus, we have converted the GUI image into feature vectors through two step abstraction, and these vector features will represent the GUI image to participate in the subsequent work of GUI cognition and action association.

### 3.2 GUI Cognition and Action Association

After acquiring the abstract representation of app static interface, the ability of dynamic behavior cognition is further trained. We propose a joint architecture of double RNN (Recurrent Neural Network) for training to obtain the test model, as shown in Figure 2. In view of the time sequence of test actions and GUI feedback during the testing process, RNN-A is used to process the action sequence, and RNN-I is used to process the GUI sequence.

Here we do not use a single app as the training object, but the app function as the training object. The reason for this is that we expect computer to learn the action intent behind the interface, not just the interface jump relationship. For example, search is a common function in apps, and although it has some appearance differences in different apps, it is still easy to identify by us humans. Then our goal is to allow the computer to cognize the search function in multiple apps through this training, rather than just targeting a certain AUT.

Therefore, in order to allow computer to find common features of functions in different interface appearances, we adopt deep learning techniques that are currently effective means to deal with such problems. In our training, based on the group of same type app, a sequence of test actions is further defined for the common functions of the apps, such as finding the search function and searching for a "The Art of Computer Programming" in different books apps. In continuous training, the computer can extract key features and correctly associate defined actions.

Specifically, in our network architecture, RNN-A is used to receive the feature vectors of the initial GUI generated by GUI abstraction, and output feasible (*action, element*) pairs that reach another target interface. At the same time, the feature vectors of the initial GUI will be delivered to RNN-I as the initial state. Then, the (*action, element*) determined by RNN-A is sent to the test execution environment to actually execute on AUT, and RNN-A obtains the response interface to generate the response interface features again; on the other hand, the (*action, element*) is passed to RNN-I, and the expected interface features are inferred from RNN-I. Finally, the actual interface features are compared with the expected interface features to check and determine the correctness of the test results.

In result judgement, we expect that the computer can judge the similarity between GUIs from a more abstract perspective rather than a simple pixel comparison. Many functions usually have similarities in the representation of GUIs of different apps, although their representation details including the position, size, and style of elements are various, such as log-in or search functions. Since we represent GUI images with feature vectors, the relative entropy [33] of the vectors is calculated to compare the similarity of GUI. Based on similar calculation results, we divide three comparison results by threshold. The first judgment is that the GUIs are almost completely similar. This situation often occurs in the homogeneous interface in the same application. For example, in the product display interface of some shopping apps, the GUI structure is exactly the same, only the product content is different. This judgment is helpful to suppress the GUI path explosion problem in the test. The second judgment is that GUIs are not completely consistent but have similarities, such as login interfaces in multiple apps. The third kind of judgment, GUIs have a big difference. The second and third judgments are conducive to the recognition of the same type of GUI and help us judge the correctness of the GUI.

Different from other model-based testing approach, in this deep learning-based modeling approach, the state of apps is represented by interface features, and the state transition is determined by the RNN-I. The action that triggers the state transition of apps is performed by the RNN-A real-time judgment.

Through the above process, we first train computer to build app function-based models. Each function is divided into different types of apps for the input and output interface and test actions of the function. And app function-based models can be converted into app type-based model for testing of AUT.

During the testing phase, the type of AUT is first obtained, the test model for this type of app applied by training phase is retrieved. Then the test actions are inferred and performed while the GUI recognition due to the test model. The test results are compared to the expected results reflected by the test model. All the test actions and judgment results are recorded to form a test report.

## 4 EVALUATION PLAN

The validation and evaluation of our vision-based automated testing will be planned from two aspects.

Firstly, evaluate the accuracy of the algorithms we introduced: (1) the accuracy of interface elements recognition through object detection; (2) the correct abstraction and judgment of interface through autoencoder and relative entropy; (3) the correctness of inferring the feasible actions and the response interface by RNN. Accuracy, precision, recall, F1-score, mAP and other evaluation metrics will be employed.

Secondly, evaluate the effectiveness of the proposed vision-based automated testing approach for mobile apps. We will use real apps (market apps from Google Play and Apple App Store, open-source apps from F-Droid and AndroTest) as test objects to compare with the state-of-the-art testing approaches, especially some highly related works [21, 25, 34, 35]. We will evaluate the performance of the following test features under different testing techniques: (1) platform dependency, whether it can effectively support cross-platform or cross-version testing; (2) permission requirement, whether it requires a higher privilege to conduct testing, such as developer privilege that may not be obtained under certain test conditions; (3) efficiency of testing, whether it has an efficient performance in the test preparation phase and execution phase; (4) supported types of testing, whether it can support various types of testing, such as functionality testing, quality of service testing, usability testing, security testing; (5) quality of testing, whether it has relatively good performance in bug detection and test coverage.

## 5 CONCLUSIONS AND FUTURE RESEARCH

Based on computer vision technology, we propose an automated mobile apps testing approach from user perspective, which provides a new idea for the current mobile apps testing automation.

At present, we have completed the preliminary realization of 3.1 *GUI Abstraction*, and 3.2 *GUI Cognition and Action Association* will be the focus of the next step. And in the follow-up, we will conduct a full experimental comparison with the current state-of-the-art testing techniques to verify and optimize our approach.

We believe that this vision-based testing approach can provide a test capability similar to manual testing to a certain extent, thereby alleviating the contradiction between the huge test requirements of mobile apps and the relatively lack of testers.

## REFERENCES

- [1] Statista. 2019. Number of apps available in leading app stores. Retrieved March 16, 2020 from <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>
- [2] Statista. 2019. Percentage of mobile device website traffic worldwide. Retrieved March 16, 2020 from <https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices>
- [3] Tramontana Porfirio, Amalfitano Domenico, Amatucci Nicola and Fasolino Anna Rita. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (March 2019), 149-201. <https://doi.org/10.1007/s11219-018-9418-6>
- [4] Kong Pingfan, Li Li, Gao Jun, Liu Kui, Bissyande Tegawende F. and Klein Jacques. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* 68, 1 (March 2019), 45-66. <https://doi.org/10.1109/TR.2018.2865733>
- [5] Espresso. Use Espresso to write concise, beautiful, and reliable Android UI tests. Retrieved March 16, 2020 from <https://developer.android.com/training/testing/espresso>
- [6] Xctest. Create and run unit tests, performance tests, and UI tests for your Xcode project. Retrieved March 16, 2020 from <https://developer.apple.com/documentation/xctest>
- [7] Appium. Automation for Apps. Retrieved March 16, 2020 from <http://appium.io>
- [8] Yeh Tom, Chang Tsung-Hsiang and Miller Robert C. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09)*, Victoria, BC, Canada. Association for Computing Machinery, 183-192. <https://doi.org/10.1145/1622176.1622213>
- [9] Eyeautomate. Visual Script Runner. Retrieved March 16, 2020 from <https://eyeautomate.com/eyeautomate/>
- [10] Gomez Lorenzo, Neamtu Julian, Azim Tanzirul and Millstein Todd. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE '13)*, San Francisco, CA, USA. IEEE, 72-81. <https://doi.org/10.1109/ICSE.2013.6606553>
- [11] Guo Jiaqi, Li Shuyue, Lou Jian-Guang, Yang Zijiang and Liu Ting. 2019. Sara: self-replay augmented record and replay for Android in industrial cases. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, Beijing, China. Association for Computing Machinery, 90-100. <https://doi.org/10.1145/3293882.3330557>
- [12] Yu Shengcheng, Fang Chunrong, Feng Yang, Zhao Wenyan and Chen Zhenyu. 2019. LIRAT: Layout and Image Recognition Driving Automated Mobile Testing of Cross-Platform. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, San Diego, CA, USA. IEEE, 1066-1069. <https://doi.org/10.1109/ASE.2019.00103>
- [13] Monkey. UI/Application Exerciser Monkey. Retrieved March 16, 2020 from <https://developer.android.com/studio/test/monkey>
- [14] Osman Mohamed S and Wasmi Hiba Ayyed. 2019. Improved Monkey Tool for Random Testing in Mobile Applications. In *Proceedings of the 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT '19)*, Amman, Jordan, Jordan. IEEE, 658-662. <https://doi.org/10.1109/JEEIT.2019.8717506>
- [15] Machiry Aravind, Tahiliani Rohan and Naik Mayur. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*, Saint Petersburg, Russia. Association for Computing Machinery, 224-234. <https://doi.org/10.1145/2491411.2491450>
- [16] Amalfitano Domenico, Fasolino Anna Rita, Tramontana Porfirio, Ta Bryan Dzung and Memon Atif M. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *Ieee Software* 32, 5 (April 2014), 53-59. <https://doi.org/10.1109/MS.2014.55>
- [17] Salihu Ibrahim-Anka, Ibrahim Roszati, Ahmed Bestoun S, Zamli Kamal Z and Usman Asmau. 2019. AMOGA: a static-dynamic model generation strategy for mobile apps testing. *IEEE Access* 7 (January 2019), 17158-17173. <https://doi.org/10.1109/ACCESS.2019.2895504>
- [18] Amalfitano Domenico, Amatucci Nicola, Fasolino Anna Rita and Tramontana Porfirio. 2015. AGRippin: a novel search based testing technique for Android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile (DeMobile '15)*, Bergamo, Italy. Association for Computing Machinery, 5-12. <https://doi.org/10.1145/2804345.2804348>
- [19] Mao Ke, Harman Mark and Jia Yue. 2016. Sapientz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*, Saarbrücken, Germany. Association for Computing Machinery, 94-105. <https://doi.org/10.1145/2931037.2931054>
- [20] Qin Xue, Zhong Hao and Wang Xiaoyin. 2019. TestMig: migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, Beijing, China. Association for Computing Machinery, 284-295. <https://doi.org/10.1145/3293882.3330575>
- [21] Behrang Farnaz and Orso Alessandro. 2019. Test migration between mobile apps with similar functionality. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, San Diego, CA, USA. IEEE, 54-65. <https://doi.org/10.1109/ASE.2019.00016>
- [22] Lin Jun-Wei, Jabbarvand Reyhaneh and Malek Sam. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, San Diego, CA, USA. IEEE, 42-53. <https://doi.org/10.1109/ASE.2019.00015>
- [23] Choi Wontae, Necula George and Sen Koushik. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.* 48, 10 (October 2013), 623-640. <https://doi.org/10.1145/2544173.2509552>
- [24] Li Yuanchun, Yang Ziyue, Guo Yao and Chen Xiangqun. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina. IEEE, 23-26. <https://doi.org/10.1109/ICSE-C.2017.8>
- [25] Li Yuanchun, Yang Ziyue, Guo Yao and Chen Xiangqun. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*, San Diego, CA, USA. IEEE, 1070-1073. <https://doi.org/10.1109/ASE.2019.00104>
- [26] Alzaylaee Mohammed K, Yerima Suleiman Y and Sezer Sakir. 2020. DL-Droid: Deep learning based android malware detection using real devices. *Computers & Security* 89 (February 2020), 101663. <https://doi.org/10.1016/j.cose.2019.101663>
- [27] Koroglu Yavuz, Sen Alper, Muslu Ozlem, Mete Yunus, Ulker Ceyda, Tanrıverdi Tolga and Dommez Yunus. 2018. QBE: QLearning-based exploration of android applications. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST '18)*, Västerås, Sweden. IEEE, 105-115. <https://doi.org/10.1109/ICST.2018.00020>
- [28] Vuong Thi Anh Tuyet and Takada Shingo. 2018. A reinforcement learning based approach to automated testing of Android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '19)*, Lake Buena Vista, FL, USA. Association for Computing Machinery, 31-37. <https://doi.org/10.1145/3278186.3278191>
- [29] Craciunescu Mihai, Mocanu Stefan, Dobre Cristian and Dobrescu Radu. 2018. Robot based automated testing procedure dedicated to mobile devices. In *Proceedings of the 2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP '18)*, Maribor, Slovenia. IEEE, 1-4. <https://doi.org/10.1109/IWSSIP.2018.8439614>
- [30] Mao Ke, Harman Mark and Jia Yue. 2017. Robotic testing of mobile apps for truly black-box automation. *Ieee Software* 34, 2 (March 2017), 11-16. <https://doi.org/10.1109/MS.2017.49>
- [31] Redmon Joseph and Farhadi Ali. 2018. Yolov3: An incremental improvement. *arXiv:1804.02767*. Retrieved from <https://arxiv.org/abs/1804.02767>
- [32] Deka Biplob, Huang Zifeng, Franzen Chad, Hibschman Joshua, Afergan Daniel, Li Yang, Nichols Jeffrey and Kumar Ranjitha. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*, Québec City, QC, Canada. Association for Computing Machinery, 845-854. <https://doi.org/10.1145/3126594.3126651>
- [33] Dizaji Kamran Ghasedi, Herandi Amirhossein, Deng Cheng, Cai Weidong and Huang Heng. 2017. Deep Clustering via Joint Convolutional Autoencoder Embedding and Relative Entropy Minimization. In *Proceedings of the 2017 IEEE International Conference on Computer Vision (ICCV '17)*, Venice, Italy. IEEE, 5747-5756. <https://doi.org/10.1109/ICCV.2017.612>
- [34] Degott Christian, Jr. Nataniel P. Borges and Zeller Andreas. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, Beijing, China. Association for Computing Machinery, 296-306. <https://doi.org/10.1145/3293882.3330569>
- [35] Borges Nataniel P., Gómez María and Zeller Andreas. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft '18)*, Gothenburg, Sweden. Association for Computing Machinery, 133-143. <https://doi.org/10.1145/3197231.3197243>

# Program-Aware Fuzzing for MQTT Applications

Luis Gustavo Araujo Rodriguez

Daniel Macêdo Batista

[luisgar@ime.usp.br](mailto:luisgar@ime.usp.br)

[batista@ime.usp.br](mailto:batista@ime.usp.br)

Department of Computer Science - University of São Paulo (USP)  
São Paulo, São Paulo, Brazil

## ABSTRACT

Over the last few years, MQTT applications have been widely exposed to vulnerabilities because of their weak protocol implementations. For our preliminary research, we conducted background studies to: (1) determine the main cause of vulnerabilities in MQTT applications; and (2) analyze existing MQTT-based testing frameworks. Our preliminary results confirm that MQTT is most susceptible to malformed packets, and its existing testing frameworks are based on blackbox fuzzing, meaning vulnerabilities are difficult and time-consuming to find. Thus, the aim of my research is to study and develop effective fuzzing strategies for the MQTT protocol, thereby contributing to the development of more robust MQTT applications in IoT and Smart Cities.

## CCS CONCEPTS

- Security and privacy → Mobile and wireless security;
- Networks → Protocol testing and verification.

## KEYWORDS

MQTT, Internet of Things, Security, Testing, Fuzzing

### ACM Reference Format:

Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. 2020. Program-Aware Fuzzing for MQTT Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3395363.3402645>

## 1 PROBLEM STATEMENT

The Internet of Things (IoT) is becoming the next step in Internet evolution [22, 24], enabling several types of devices to interact through communication protocols. Among IoT protocols, the Message Queuing Telemetry Transport (MQTT) protocol is considered the best, offering lightweight-messaging and low-bandwidth consumption [4, 5, 16, 30]. Over the last few years, MQTT applications have increased drastically [15, 18]. In fact, MQTT currently ranks as the most popular publish-subscribe protocol [11]. Moreover, MQTT

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3402645>

is the most widely-used *IoT protocol* [11], standardized by ISO/IEC 20922 and OASIS.

Currently, research on MQTT security is underdeveloped [15]. In fact, MQTT applications have been widely criticized over the last few years for their lack of security [21] and faulty implementations in real-world environments [2, 3, 7, 18, 28]. Considering this issue, we decided to conduct studies to answer two main background questions: (1) *What are the most common types of flaws in these implementations?*; and (2) *What testing frameworks have been proposed to mitigate these flaws?*

In order to answer the first question, we conducted a manual study to determine the root cause of vulnerabilities. All MQTT-related vulnerabilities (As of April 22nd, 2020) were collected from the National Vulnerability Database (NVD), which is the most widely-used exploit repository [13]. Based on our findings presented in Figure 1, thirty-seven vulnerabilities (71.15%) are triggered by malformed MQTT control packets.

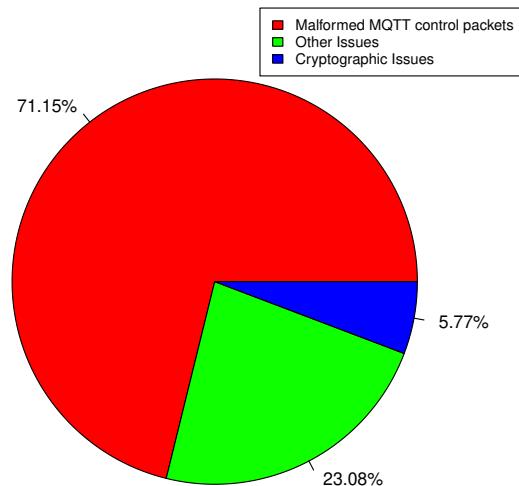


Figure 1: Causes of vulnerabilities in MQTT applications

Malformed control packets could allow information disclosure; remote code execution; and denial of service, thereby hindering confidentiality, integrity, and availability respectively [20]. In fact, a stack overflow vulnerability in MQTT (CVE-2019-11779)<sup>1</sup> was recently discovered by sending a crafted subscribe packet.

Similar to the research by Rodriguez et al. [25], we've analyzed all vulnerability disclosures related to MQTT from NVD. Based on our findings, vulnerabilities triggered by malformed packets

<sup>1</sup><https://nvd.nist.gov/vuln/detail/CVE-2019-11779>. Accessed on Apr 30th, 2020

have higher disclosure delays than their counterparts, hindering immediate measures against cyberattacks. In addition to disclosure delays, deploying and applying patches is usually complex in MQTT environments. For example, open source home automation software such as *HomeAssistant* lack auto-update functionality, meaning MQTT flaws are rarely patched. Moreover, popular home automation devices such as *Sonoff* are vulnerable to malformed MQTT packets [1], and require manual intervention to update as well.

## 2 RELATED WORK

Testing MQTT applications with effective mechanisms can mitigate these aforementioned issues [7, 27–29]. Thus, in order to answer the second background question (*What testing frameworks have been proposed to mitigate these flaws?*), we conducted a literature review of existing security-testing frameworks for MQTT. Since MQTT is most susceptible to vulnerabilities triggered by malformed packets, *fuzz testing* [19] can play a key role in mitigating this issue. In fact, fuzz testing is considered one of the most promising methods for discovering vulnerabilities in IoT [17]. Thus, the following subsections focus specifically on existing fuzzing frameworks for MQTT. The subsections are organized by the year the initial versions of these frameworks were released.

### 2.1 2015 or Earlier

Defensics<sup>2</sup> is a hybrid blackbox fuzzer, developed by Synopsys, for several protocols, including MQTT. Defensics offers over 250 ready-made test cases; an SDK for custom-made test cases; and a traffic capture fuzzer to generate new test cases. A process monitor and reports are provided for analysis. Although Defensics provides several tools for security-testing, it is not open-source, thereby preventing a higher user adoption.

F-Secure Corporation [12] propose *mqtt-fuzz*, a blackbox mutation-based fuzzer for the MQTT protocol. Its aim is to reduce effort and offer simplicity to developers, thereby lacking complex protocol processing. Although *mqtt-fuzz* is open-source, it requires considerable amount of test cases to reach deep protocol states.

### 2.2 2017

Anantharaman et al. [3] construct a Finite State Machine (FSM) based on specifications of the MQTT protocol. An input language and parsers were defined and developed for each state respectively. A blackbox generation-based fuzzer was developed to test these parsers. The fuzzer generates test cases based on the input language, meaning it sends correctly-formed packets to the System Under Test (SUT). Rather than using the fuzzer to discover vulnerabilities, this research uses it to test the semantic correctness of the messages.

### 2.3 2018

Hernández Ramos et al. [15] propose a mutation-based, blackbox fuzzer for the MQTT protocol. The aim of this research is to reduce effort when verifying security of MQTT applications. The architecture of the fuzzer mainly consists of a network sniffer, a template-generator, and the fuzzer itself. The sniffer retrieves

<sup>2</sup><https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html>. Accessed on November 17th, 2019

network packets between the client and the broker. The template-generator generates and exports templates based on the network packets. From these templates, the user manually marks the fields that will be fuzzed. The fuzzer then mutates these fields by using test cases provided by the user or generating random inputs automatically. The experiments were done with two MQTT implementations: Mosquitto and Moquette. A total of three vulnerabilities were discovered, being mostly denial of service. Although the fuzzer had low CPU consumption, it lacks coverage feedback.

Eclipse Foundation [10] proposes a fuzz testing framework for the MQTT protocol. The fuzzer acts as a man-in-the-middle, mutating network packets between the client and the server. It is a mutation-based blackbox fuzzer, requiring valid templates to generate the test cases. Similar to the fuzzer proposed by Hernández Ramos et al. [15], the user manually selects the messages that will be fuzzed. The developers warn that basic random generators are used to mutate the messages.

Although fuzzing frameworks by Hernández Ramos et al. [15] and the Eclipse Foundation [10] have low-time complexity, they act as a proxy and thus lack support for stateful fuzzing [8, 15].

### 2.4 2019

Palmieri et al. [21] propose *MQTTSA*, a penetration testing framework for MQTT. *MQTTSA* detects potential vulnerabilities and generates a report offering suggestions to mitigate these issues. *MQTTSA* consists of three penetration-testing mechanisms: authentication bruteforcing, data tampering, and denial of service. Data tampering is based on a mutation-based blackbox fuzzer. Experiments were done with brokers found online and five different deployments of Mosquitto. Approximately 60% of brokers found online lack confidentiality and integrity. Although Mosquitto proves to be secure when configured correctly, this research recommends complementing *MQTTSA* with more effective fuzzing techniques or using F-Secure's fuzzer [12] for better testing.

### 2.5 Limitations of Existing MQTT Fuzzers

We classified each MQTT fuzzer into three categories (Table 1): understanding the target program; aware of the input structure; and input generation. Analyzing the current state-of-art reveals that existing MQTT fuzzers have several limitations. First, existing MQTT-based testing frameworks are based on blackbox fuzzing, meaning they require considerable amount of test cases to reach deep protocol states [21]. Currently, greybox and whitebox MQTT fuzzers are nonexistent. Second, these blackbox fuzzers discard coverage-increasing inputs during the fuzzing campaign. This means that many inputs could traverse the same path. As a result, vulnerabilities are difficult and time-consuming to find. Third, the probability of false negatives is high because of the lack of program analysis. In addition to the aforementioned fuzzers, popular platforms such as *Peach Fuzzer*<sup>3</sup>, *F-Interop*<sup>4</sup>, and *American Fuzzy Lop*<sup>5</sup> currently lack support for MQTT. Because of these limitations and lack of support, recent research papers have expressed interest in a *smart MQTT fuzzer* for more effective testing [3, 21].

<sup>3</sup><https://www.peach.tech/products/peach-fuzzer/>. Accessed on August 7th, 2019

<sup>4</sup><https://www.f-interop.eu/>. Accessed on June 8th, 2019

<sup>5</sup><https://github.com/google/AFL>

**Table 1: Classification of existing MQTT fuzzers**

Existing Frameworks	Understanding target program			Aware of input structure		Input generation		Open Source
	Blackbox	Greybox	Whitebox	Smart	Naive	Mutation-Based	Generation-Based	
Synopsys	✓			✓		✓	✓	
F-Secure Corporation, 2015 [12]	✓			✓		✓		✓
Anantharaman et al. (2017) [3]	✓			✓			✓	
Hernandez et al. (2018) [15]	✓			✓		✓		✓
Eclipse Foundation, 2018 [10]	✓			✓		✓		✓
Palmieri et al. (2019) [21]	✓			✓		✓		✓

### 3 PROPOSED RESEARCH

Considering our studies presented in Sections 1 and 2, the aim of the Ph.D. research is to study and develop effective strategies for fuzz testing the MQTT protocol. More specifically, we want to address limitations of existing MQTT fuzzers by using *program analysis*, thereby offering developers a more modern fuzzer than previous works in the literature. Program analysis can be performed through greybox or whitebox approaches. Our goal is to develop a fuzzing framework specifically adapted to MQTT's characteristics.

Analyzing MQTT's vulnerabilities and existing frameworks was crucial for constructing our hypothesis, which is:

*When using program-aware fuzzing, efficiency is considerably higher than that of existing MQTT fuzzers.*

The research questions that will be used to validate our hypothesis are as follows.

**RQ1** How efficient are existing fuzzing frameworks for MQTT?

**RQ2** How efficient are program-aware fuzzing techniques for MQTT?

**RQ3** What are the characteristics of MQTT vulnerabilities that are revealed by means of fuzz testing?

This research will involve three main efforts: (1) an empirical study of existing MQTT-based testing frameworks; (2) the development of program-aware approaches for fuzzing MQTT applications; and (3) an evaluation of different fuzzing techniques in virtual and real-world environments, determining the most suitable approach for MQTT. The main results expected from this research are: (1) improved understanding of existing MQTT fuzzers; (2) effective strategies for fuzzing MQTT; and (3) robust MQTT applications in IoT and Smart Cities. This research will provide two types of contributions:

**Scientific Contribution:** The methodology used for this research will support future developments of testing tools either for MQTT, publish-subscribe protocols, or IoT in general.

**Technical Contribution:** To the best of our knowledge, no greybox or whitebox fuzzer exists for MQTT. Thus, this research will provide the *first program-aware fuzzer for MQTT*. The tools developed for this research will be open-source, allowing developers to test their MQTT applications and improve or build on top of our work for future research.

### 4 METHODOLOGY AND EVALUATION PLANS

An *MQTT broker* needs to be heavily robust against cyberattacks because it handles requests from publishers and subscribers [6, 14].

Since the broker is considered to be the main component of the publish-subscribe model, this research will focus on testing broker-side implementations of MQTT such as *Mosquitto*.

Our testbed consists of two main components: the fuzzer and the MQTT broker, simulating an attacker and a target system respectively. Both components were configured to communicate with each other on a private network. The goal is to determine the most suitable fuzzing technique for MQTT, whether it be blackbox or program-aware approaches. Test effectiveness or efficiency of each fuzzer (RQ1 and RQ2) will be measured in terms of code coverage; state exploration; unique crashes; and execution times. The following subsections explain how we attempt to tackle each research question.

#### 4.1 RQ1: How efficient are existing fuzzing frameworks for MQTT?

The current state-of-art (Table 1) shows little evidence of the effectiveness of existing fuzzing frameworks for MQTT. Thus, we are currently conducting experiments to evaluate these tools individually. The goal is to analyze their fuzzing efficiency, which will then serve as a baseline to compare with program-aware approaches.

**4.1.1 Ongoing Experiments.** We are currently conducting experiments with F-Secure's MQTT fuzzer [12] to evaluate its code coverage and vulnerability detection capabilities. Thus far, we tested the most recent versions of Mosquitto (1.6.0 - 1.6.9), using *Gcov* and *AddressSanitizer* as our instrumentation tools.

**4.1.2 Preliminary Results.** F-Secure's fuzzer injects malformed packets into each version of Mosquitto for twenty-four hours. Approximately ten million packets were exchanged between the fuzzer and the broker during the campaign. However, in most cases (Versions 1.6.0, 1.6.1, 1.6.3, 1.6.4, and 1.6.5 of Mosquitto) only 32% of the code was covered by the fuzzer. In three cases (Versions 1.6.2, 1.6.6, and 1.6.8) only 33% of the code was covered and in two cases (Versions 1.6.7 and 1.6.9) only 31% of the code was covered. This means that most test cases traversed through the same path, mainly because of the lack of coverage feedback. Moreover, neither existing nor non-existing vulnerabilities were triggered during the fuzzing campaign. We've also conducted a few experiments with *MQTTSA*'s fuzzer. It managed to cover at most 25% of *Mosquitto*'s source code, further highlighting the inefficiency of public MQTT fuzzers.

We plan to continue performing experiments with other open-source fuzzing frameworks (Table 1) to evaluate their efficiency.

## 4.2 RQ2: How efficient are program-aware fuzzing techniques for MQTT?

Preliminary results from **RQ1** suggest the necessity of program-aware approaches for fuzzing MQTT applications. With the insights from **RQ1**, we will develop greybox or whitebox approaches that attempt to mitigate limitations of existing frameworks. Since no program-aware fuzzer exists for MQTT, there is a wide range of techniques that can be considered for this research such as guided-fuzzing or symbolic execution.

At the time of writing, we plan to focus first on greybox approaches for improving fuzzing efficiency. Whitebox techniques will be considered depending on our research schedule. Greybox approaches were selected because they combine advantages of both blackbox and whitebox fuzzers, using lightweight instrumentation with minimal overhead. More specifically, we plan on using *Genetic or evolutionary algorithms*, for traversing unexplored or specific regions of the code during the fuzzing campaign. Figure 2 presents an initial flow diagram of each component that will be considered for developing our program-aware fuzzer.

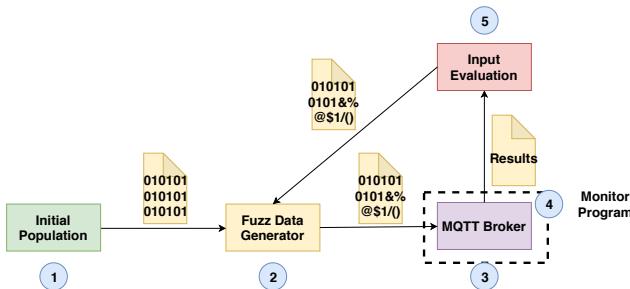


Figure 2: Architecture of a program-aware fuzzer for MQTT

**4.2.1 Initial Population.** The initial test cases will be generated by capturing valid packets using tools such as Wireshark or Scapy. This approach has been used in several frameworks for MQTT [12, 15]. However, preliminary results from **RQ1** suggest that existing MQTT fuzzers may have difficulties in triggering existing or non-existing vulnerabilities. A more effective approach would be to generate initial test cases based on existing vulnerabilities. Thus, we plan to build a dataset of malicious inputs from existing CVE reports, which will be used as *seeds* to trigger potential flaws.

**4.2.2 Fuzz Data Generator.** Preliminary results from **RQ1** suggest that existing MQTT fuzzers achieve low state coverage. Thus, our fuzzer will interact directly with the target system, rather than acting as a man-in-the-middle. This approach was chosen in order to offer a more flexible framework, capable of generating test cases based on existing vulnerabilities, a grammar, or whitebox approaches. Best practices for mutating inputs will also be studied.

**4.2.3 MQTT Broker.** Open-source brokers such as *Mosquitto*; *MQTtdu*; *Emqttd*; and *RabbitMQ* will be considered to enable program-aware approaches, and thus improve the accuracy of the tests [28]. These brokers were selected because of their popularity and inconsistencies with the protocol standard [9, 15, 20].

**4.2.4 Monitor Program and Input Evaluation.** Once a test case is selected and injected to the target system, the most effective test cases will be kept for future iterations. Since preliminary results from **RQ1** reveal that most test cases traverse through the same path, the criteria (or *fitness function*) for selecting test cases will be based on code coverage or state exploration. We plan to perform the following activities: (1) study and complement our approach with effective techniques such as stateful fuzzing, which has been successful for network protocols [8, 23]; (2) conduct a qualitative analysis to receive feedback from developers on our fuzzing approach and important metrics to test MQTT applications; (3) use input minimization [26], by reducing and keeping the most effective test cases; (4) select efficient evolutionary algorithms for generating test cases; (5) test MQTT applications deployed in virtual and real-world environments; and (6) analyze experiment results and determine the most effective approach for testing MQTT applications.

## 4.3 RQ3: What are the characteristics of MQTT vulnerabilities that are revealed by means of fuzz testing?

We plan to analyze the characteristics of vulnerabilities found during our experiments. This analysis could allow us to create a taxonomy to classify security vulnerabilities in MQTT applications.

## 5 CONCLUSIONS

MQTT is currently being integrated into several IoT-based applications, such as home automation systems; health monitoring systems; and intelligent transportation systems. Considering these scenarios, it is important to mitigate potential attacks and increase reliability of MQTT applications. However, MQTT applications have been widely criticized over the last few years for their weak protocol implementations. Our background studies confirm that MQTT is most susceptible to malformed packets, hence the importance of fuzz testing. Existing MQTT fuzzers are based on blackbox testing, lacking any type of program analysis, thereby offering difficulty to discover vulnerabilities. Thus, the aim of this research is to study and develop effective strategies for fuzz testing the MQTT protocol. We are currently performing experiments with existing frameworks to evaluate their efficiency, which will be crucial when developing program-aware approaches for fuzzing MQTT applications. Our program-aware fuzzer will be open-source, allowing developers to test their MQTT applications effectively, thereby deploying more robust MQTT applications in IoT and Smart Cities.

## ACKNOWLEDGMENTS

We would like to thank Professor Maurício Aniche from TU Delft, and Professor Paulo Lício de Geus from UNICAMP for their suggestions to improve our research. This paper would not have been possible without their support. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9. This research is also funded by FAPESP proc. 18/22979-2 and CAPES.

## REFERENCES

- [1] Daniel Abeles and Moshe Zioni. 2019. MQTT-PWN Documentation. <https://buildmedia.readthedocs.org/media/pdf/mqtt-pwn/latest/mqtt-pwn.pdf>. [Online; accessed 17-September-2019].
- [2] Khalid Alghamdi, Ali Alqazzaz, Anyi Liu, and Hua Ming. 2018. Iotverif: An automated tool to verify SSL/TLS certificate validation in android MQTT client applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 95–102. <https://doi.org/10.1145/3176258.3176334>
- [3] P. Anantharaman, M. Locasto, G. F. Ciocarlie, and U. Lindqvist. 2017. Building Hardened Internet-of-Things Clients with Language-Theoretic Security. In *Proceedings of the IEEE Security and Privacy Workshops*. 120–126. <https://doi.org/10.1109/SPW.2017.36>
- [4] S. Andy, B. Rahardjo, and B. Hanindhito. 2017. Attack scenarios and security analysis of MQTT communication protocol in IoT system. In *Proceedings of the International Conference on Electrical Engineering, Computer Science and Informatics*. 1–6. <https://doi.org/10.1109/ECCSL.2017.8239179>
- [5] Joseph Jose Anthraper and Joseph Kotak. 2017. Security, Privacy and Forensic Concern of MQTT Protocol. In *Proceedings of the International Conference on Sustainable Computing in Science, Technology and Management*. 1–8. <https://doi.org/10.2139/ssrn.3355193>
- [6] João Antunes, Nuno Neves, Miguel Correia, Paulo Verissimo, and Rui Neves. 2010. Vulnerability discovery with attack injection. *IEEE Transactions on Software Engineering* 36, 3 (2010), 357–370. <https://doi.org/10.1109/TSE.2009.91>
- [7] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, Xiaofeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTfuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Network and Distributed Systems Security Symposium*. <https://doi.org/10.14722/ndss.2018.23159>
- [8] Yurong Chen, Tian Ian, and Guru Venkataramani. 2019. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom). Association for Computing Machinery, New York, NY, USA, 17–23. <https://doi.org/10.1145/3338502.3359762>
- [9] D. L. de Oliveira, A. F. da S. Veloso, J. V. V. Sobral, R. A. L. Rabélo, J. J. P. C. Rodrigues, and P. Solic. 2019. Performance Evaluation of MQTT Brokers in the Internet of Things for Smart Cities. In *Proceedings of the International Conference on Smart and Sustainable Technologies*. 1–6. <https://doi.org/10.23919/SpliTech.2019.8783166>
- [10] Eclipse Foundation. 2018. Eclipse IoT-Testware. [https://iottestware.readthedocs.io/en/development/smarter\\_fuzzer.html](https://iottestware.readthedocs.io/en/development/smarter_fuzzer.html). [Online; accessed 17-October-2019].
- [11] Eclipse Foundation. 2019. IoT Developer Survey. <https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf>. [Online; accessed 15-June-2019].
- [12] F-Secure Corporation. 2015. A simple fuzzer for the MQTT protocol. [https://github.com/F-Secure/mqtt\\_fuzz](https://github.com/F-Secure/mqtt_fuzz). [Online; accessed 16-September-2019].
- [13] Ming Fang and Munawar Hafiz. 2014. Discovering Buffer Overflow Vulnerabilities in the Wild: An Empirical Study. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Article 23, 10 pages. <https://doi.org/10.1145/2652524.2652533>
- [14] S. N. Firdous, Z. Baig, C. Valli, and A. Ibrahim. 2017. Modelling and Evaluation of Malicious Attacks against the IoT MQTT Protocol. In *Proceedings of the IEEE International Conference on Internet of Things and the IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*. 748–755. <https://doi.org/10.1109/iThings-GreenCom-CPSCoSmartData.2017.115>
- [15] Santiago Hernández Ramos, M. Teresa Villalba, and Raquel Lacuesta. 2018. MQTT Security: A Novel Fuzzing Approach. *Wireless Communications and Mobile Computing* 2018 (2018), 1–11. <https://doi.org/10.1155/2018/8261746>
- [16] Rob Kitchin and Martin Dodge. 2019. The (In)Security of Smart Cities: Vulnerabilities, Risks, Mitigation, and Prevention. *Journal of Urban Technology* 26, 2 (2019). <https://doi.org/10.1080/10630732.2017.1408002>
- [17] Jian-Zhen Luo, Chun Shan, Jun Cai, and Yan Liu. 2018. IoT Application-Layer Protocol Vulnerability Detection using Reverse Engineering. *Symmetry* 10, 11 (2018), 561. <https://doi.org/10.3390/sym10110561>
- [18] Federico Maggi, Rainer Vosseler, and Davide Quarta. 2018. The Fragility of Industrial IoT's Data Backbone. [https://documents.trendmicro.com/assets/white\\_papers/wp-the-fragility-of-industrial-IoTs-data-backbone.pdf](https://documents.trendmicro.com/assets/white_papers/wp-the-fragility-of-industrial-IoTs-data-backbone.pdf)
- [19] Valentin Jean Marie Manés, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019).
- [20] Kristiyian Mladenov, Stijn Van Wissen, Chris Mavrakis, and KPMG Cyber. 2017. *Formal verification of the implementation of the MQTT protocol in IoT devices*. Master's dissertation. University of Amsterdam.
- [21] Andrea Palmieri, Paolo Prem, Silvio Ranise, Umberto Morelli, and Tahir Ahmad. 2019. MQTTSA : A Tool for Automatically Assisting the Secure Deployments of MQTT brokers. In *IEEE World Congress on Services*, Vol. 2642-939X. IEEE, 47–53. <https://doi.org/10.1109/SERVICES.2019.00023>
- [22] Yusuf Perwej, Majzoob Omer, Osama Sheta, Hani Harb, and Mohammed Adrees. 2019. The Future of Internet of Things (IoT) and Its Empowering Technology. *International Journal of Engineering Science and Computing* Volume 9 (March 2019), 20192 – 20203.
- [23] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*.
- [24] Pedro Martins Pontes, Bruno Lima, and João Pascoal Faria. 2018. Izinto: A Pattern-Based IoT Testing Framework. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 125–131. <https://doi.org/10.1145/3236454.3236511>
- [25] Luis Gustavo Araujo Rodriguez, Julia Selvatici Trazzi, Victor Fossalaza, Rodrigo Campiolo, and Daniel Macêdo Batista. 2018. Analysis of Vulnerability Disclosure Delays from the National Vulnerability Database. In *Proceedings of the Workshop on CyberSecurity in Connected Devices in the Brazilian Symposium on Computer Networks and Distributed Systems*. <https://portaldeconteudo.sbc.org.br/index.php/wscdc/article/view/2394>
- [26] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. *arXiv* (2019).
- [27] M. Schiefer. 2015. Smart Home Definition and Security Threats. In *Proceedings of the International Conference on IT Security Incident Management IT Forensics*. 114–118. <https://doi.org/10.1109/IMF.2015.17>
- [28] Synopsis. 2017. State of Fuzzing. <https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/state-of-fuzzing-2017.pdf>. [Online; accessed 15-June-2019].
- [29] C. Săndescu, O. Grigorescu, R. Rughiniș, R. Deaconescu, and M. Calin. 2018. Why IoT security is failing. The Need of a Test Driven Security Approach. In *Proceedings of the International Conference: Networking in Education and Research*. 1–6. <https://doi.org/10.1109/ROEDUNET.2018.8514135>
- [30] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi. 2017. Internet of Things: Survey and open issues of MQTT protocol. In *Proceedings of the International Conference on Engineering MIS*. 1–6. <https://doi.org/10.1109/ICEMIS.2017.8273112>

# Automatic Support for the Identification of Infeasible Testing Requirements

João Choma Neto

joaochoma@usp.br

Institute of Mathematics and Computer Sciences, University of São Paulo

São Carlos, São Paulo, Brazil

## ABSTRACT

Software testing activity is imperative to improve software quality. However, finding a set of test cases satisfies a given test criterion, is not a trivial task because the overall input domain is very large, and different test sets can be derived, with different effectiveness. In the context of structural testing, the non-executability is a feature present in most programs, increasing cost and effort of testing activity. When concurrent programs are tested, new challenges arise, mainly related to the non-determinism. Non-determinism can result in different possible test outputs for the same test input, which makes the problem of non-executability more complex, requiring treatment. In this sense, our project intends to define an approach to support automatic identification of infeasible testing requirements. Hence, this proposal aims to identify properties which cause infeasible testing requirements and automate their application. Due to complexity of the problem, we will apply search-based algorithms in the automation of concurrent and sequential programs treatment.

## CCS CONCEPTS

- Software and its engineering → Software testing and debugging; Empirical software validation; Search-based software engineering.

## KEYWORDS

Software Testing, Structural Testing, Infeasible Path Problem, Search Based Software Testing

### ACM Reference Format:

João Choma Neto. 2020. Automatic Support for the Identification of Infeasible Testing Requirements. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3395363.3402646>

## 1 INTRODUCTION

Testing techniques and criteria have been proposed to systematize the testing activity. Among them, structural testing, also called white-box test, uses information about internal structure of system

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8008-9/20/07...\$15.00

<https://doi.org/10.1145/3395363.3402646>

to derive test cases [20]. Each technique presents a set of testing criteria, which systematize the test activity, assisting in two aspects: 1) in the selection and/or generation of test data, and 2) in the decision when this activity can be completed.

Structural testing criteria can use a measure of coverage of testing requirements for evaluating the testing activity progress, helping to decide if tests are enough. This measure compares how many testing requirements were covered by test data set used. The process of applying test criteria is faced with the problem of non-executability also known as the problem of infeasible paths. This problem is recognized as a limitation of testing activity and it depends directly on the tester's skills [6]. The problem of non-executability extends to any testing requirements, so we will focus on infeasible testing requirements problem. Most software systems have infeasible testing requirements [4, 10] what negatively interfere in coverage analysis and evolution of the testing activity.

The infeasible testing requirements problem affects the effort of applying structural testing criteria [8, 22, 30, 32]. This raises the cost of generating test cases what may prevent the program from achieving satisfactory levels of quality during static analysis. One solution to this problem would be to remove infeasible paths from testing process, leading to improvements in code coverage results, although such action is not trivial [18].

In other perspective, concurrent programming is an essential paradigm to reduce computational time in many application domains (e.g. for instance, web servers). A concurrent program is composed of concurrent (or parallel) processes (or threads) which interact to solve a complex problem. This interaction may occur synchronously or not, in which these processes may or not compete for the same computational resources [23]. Concurrent programs are more complex compared to sequential programs due to the following features: [9, 23]: non-deterministic behavior, large number of synchronization sequences, new type of defects, and high complexity for automatic generation of test data. The problem of infeasible testing requirements is aggravated in concurrent programs due to non-determinism in the generation of synchronization sequences.

Although detection of all infeasible testing requirements is an intractable problem [21], many approaches have been proposed to support partial identification of infeasible testing requirements. Among the techniques developed are: symbolic evaluations, manual identification in control and data flow graphs, branch correlation and polymorphic call [15]. Other approaches have been developed based on heuristics [4, 17] and others have proposed code patterns for detecting of infeasible paths [21]. Research on infeasible testing requirements in concurrent programs focuses on eliminating its occurrence, for example, with the proposition of Reachability Testing [16], but with a high application cost. Other works explore

static analysis and symbolic executions [5, 27] and data stream analysis and execution monitoring [1].

A new approach using search algorithms has emerged to address problems considered difficult to solve in Software Engineering. Since 1976, the number of publications involving software testing using search-based algorithms have increased, giving rise to the term Search Based Software Testing (SBST) [19]. The SBST is used to solve or minimize problems in the context of: functional testing, structural testing, integration testing, regression testing, stress testing, mutation testing, test prioritization, model-based testing and exception tests [19]. In this context, a gap regarding the use of SBST to address the problem of infeasible synchronization sequences in concurrent programs is observed. Although there are approaches which address the infeasible testing requirements problem, there are no studies that address the infeasible testing requirements problem in concurrent programs, using SBST techniques. The main benefit of this proposal is allows automatic identification of infeasible testing requirements, reducing cost of test data generation process.

Manual or automated detection of infeasible testing requirements prior to test data generation presents the following challenges: a) eliminating all infeasible testing requirements is a undecidable problem; b) the infeasible testing requirements problem is present in structural testing of sequential programs and is intensified in concurrent programs context, due to synchronization sequences; c) concurrent programs exhibit non-deterministic behavior, which produces a large set of possible synchronization sequences among processes; d) the computational cost of identifying infeasible paths at runtime is high; e) the cost of static identification of infeasible paths satisfactorily reduces the computational cost of generating test data; f) the search approach, often used for automatic test data generation, can be refined to identify infeasible paths; g) human perception hitherto used to detect infeasible paths can directly assist in the process of automatically detecting infeasible paths.

Previous works related to identification of infeasible testing requirements are input test dependencies, what favors using automatic test data generation as a means of identifying infeasible testing requirements. This approach demands a high computational cost and its effectiveness is not always achieved [4, 17]. In other approaches, infeasible testing requirements identification properties are applied manually and empirically. Despite using tester's experience, this approach loses its effectiveness due to cognitive exhaustion of the manual identification process [28].

We have identified a research gap focused on the need to define a set of properties for identifying automatically infeasible paths. Taking into account observations and the research gap, we consider that relevant properties should be automatically applied and without depending on the input data.

Studies indicate that infeasible paths are present in all software systems, impacting software testing activity, in particular, in analysis of test coverage and in generation of test data [8, 32]. The full automaton of the identification of infeasible testing requirements is undecidable problem [6]. The automatic determination of infeasible paths, although desired, is still a challenge, and use of search-based algorithms can be useful because it could mitigate the non-determinism of synchronization sequence formation in concurrent programs.

This paper presents a doctoral proposal which aims to address the problem of detecting infeasible testing requirements in structural testing of concurrent and sequential programs, with the support of search-based algorithms. For this, the paper is organized as follows: Section 2 presents motivations and objectives, Section 3 presents the proposal development methodology and progress and, finally, the Section 4 presents the expected results.

## 2 RESEARCH OBJECTIVES

The aspects which motivate the development of this doctoral project are: (i) the infeasible testing requirements problem is present in all programs and, in the context of concurrent programs, this problem is intensified due to existing communication and synchronization; (ii) there is no current approach that is capable of properly detecting the infeasible testing requirements, it is undecidable problem and its solution is not found in polynomial time; (iii) the detection of infeasible paths before the test data generation makes possible to drastically reduce the cost of this activity, allowing a more effective generation of test data; (iv) search-based algorithms, initially proposed to support the automatic generation of test data, can be adapted to identify infeasible paths; (v) human perception, previously used to manually detect infeasible testing requirements, can directly assist in the automatic detection process; (vi) concurrent programs have non-deterministic behavior which produces a high set of possible sequences of synchronization among processes, increasing the search space favoring the application of search-based algorithms.

Considering this scenario, the objective of this doctoral project is to investigate the application of search-based algorithms to support the identification of infeasible testing requirements. Initially, the proposal will be applied on sequential programs, which will serve as a basis to instantiate the proposal for concurrent programs. In this sense, this project intends to answer the following research question, **Research Question:** *How can search-based algorithms be applied to support the identification of infeasible testing requirements?*. Based on the main research question, we identified other secondary questions that can be answered with our project.

**Question 1:** *Which properties for identification of infeasible testing requirements can be automatically applied?*, for the reason that literature does not present any process that automates the identification of infeasible testing requirements.

**Question 2:** *Which search-based algorithms resources are efficient to help the problem of infeasible testing requirements?*, for the reason that of non-determinism, computational, and cost to for automatic test data generation process, for concurrent programs.

**Question 3:** *Does the incorporation of human perceptions assist in the automatic identification of infeasible testing requirements?*, for the reason of non-determinism in synchronization sequences, and computational cost to automatic generation of test data, in concurrent programs.

**Question 4:** *Which benefits are achieved with the application of properties for identification of infeasible testing requirements in the structural testing activity?*, due to development of the doctoral project will directly benefit the software testing domain, specifically the structural test.

### 3 METHODOLOGY AND PROGRESS

To answer the research questions and build the proposed approach, we elaborated the following methodology:

- Step 1: Set properties to identify infeasible paths for concurrent and sequential programs.
- Step 2: Implement the algorithm for automatically applying properties.
- Step 3: Formalize the identification properties of infeasible testing requirements for concurrent programs.
- Step 4: Automate the properties developed in Step 3.
- Step 5: Use search-based algorithms to optimize the automation algorithm in the context of concurrent programs.
- Step 6: Incorporation of human perception in the process of identifying infeasible testing requirements.

In Step 1, we have developed a systematic mapping to identify studies which investigate the problem of non-executability in concurrent and sequential programs. We use Snowballing systematic methodology, which uses references and citations from seminal and highly cited studies to gather relevant studies to answer the research questions [24, 31]. The systematic mapping artifacts can be found at GitHub<sup>1</sup>. Among the results achieved in systematic mapping, we identify five properties that identify infeasible testing requirements, and that can be applied automatically:

- **P1 - Assignment of a constant value to a variable** [12, 29]: A constant assigned to a variable may imply a specific conditional direction causing an infeasible testing requirements.
- **P2 - Opposite Predicates - Equal Predicates** [21, 30]: a dependency between opposite or equal predicates in one path may lead to infeasible testing requirements due to the conditional assessment of one predicate interfering with the evaluation of following predicate.
- **P3 - Correlation between conditional statements** [3, 22]: correlation between some conditional statements along the can cause infeasible testing requirements.
- **P4 - Change of the definition of a variable during the analyzed path** [21]: Changing the definition of a variable can generate infeasible testing requirements in the presence of loops.
- **P5 - Dead codes** [2]: Logically inconsistent predicates related to dead codes cause infeasible testing requirements due to non-execution of the code element.

Some facts were obtained by mapping study: (i) source code properties that reveal potential infeasible testing requirements; (ii) which properties do not require input data to reveal infeasible requirements; (iii) the application of the properties can be automated, similar to automatic generation of input data [4, 7, 11, 13, 14, 22, 25, 26]; and, (iv) the application of property P4 needs help of tester to make decision about potential infeasible requirements.

Still in Step 1, we have executed an empirical study for evaluating the properties, manually applying in a set of sequential programs. We observed that properties reveal infeasible testing requirements

without the need input data and can be applied automatically. Study artifacts can be found at GitHub<sup>2</sup>.

An automated approach contributes to reduce tester's effort and decrease the cost of generating input data. This reduction is caused because infeasible requirements will no longer be considered in the structural testing. In spite of reducing tester's effort, the automation will not replace it once there are properties which need the tester's contribution to increase its effectiveness. Hence, we realized that including human perception in automation process, benefits to the treatment of the infeasible testing requirements problem can be achieved. The property identification algorithm will not be able to make decision in all cases, so on that occasion the tester will decide whether the testing requirements is feasible or infeasible. This procedure can later be replaced by a machine learning algorithm.

To start the development of Step 2, we outlined in, Figure 1, the main stages of the algorithm which automates the identification of infeasible testing requirements. Our goal is output generated by the approach will be a refined set of testing requirements that will reduce the cost of generating test data and consequently reduce the cost of testing activity.

**Stage 1 - Instrumented code generation.** This stage receives as input the code to be tested. This stage will be responsible for instrumenting the code, so that the control flow graphs are constructed.

**Stage 2 - Selection and Application of Test Criteria.** This stage will apply testing criteria to the instrumented code received as input. As output, it will generate a set of testing requirements. The testing criteria used will be those which can properties, e.g. data flow testing favoring Property P4.

**Stage 3 - Application of properties.** This stage will apply the identification properties to testing requirements generated in Stage 2. The application process will signal which testing requirements are infeasible or potentially infeasible. As a result of this stage, a set of feasible testing requirements and a set of infeasible testing requirements will be generated. The feasible set of testing requirements will be used in Application Test stage, while the infeasible set will be ignored.

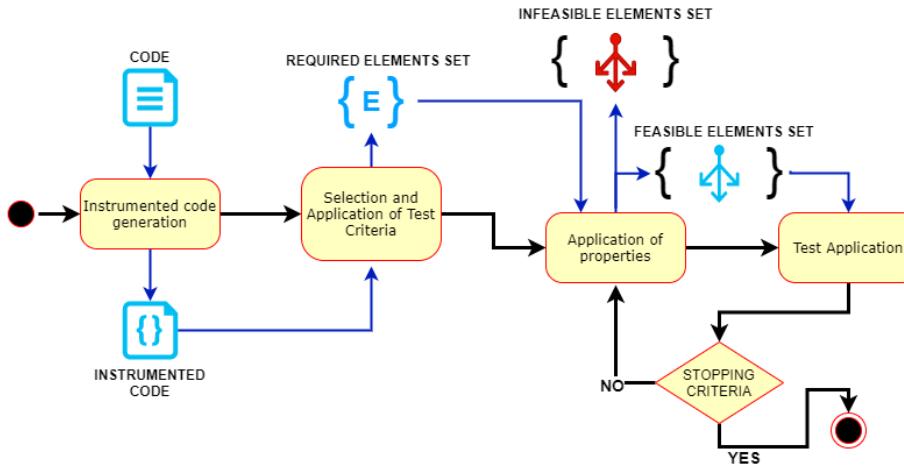
**Stage 4 - Test Application.** This stage will cover the set of testing requirements received from Stage 3. The output will be the coverage rate that will be the criterion for stopping the process. If the coverage rate is satisfactory, then the process will be completed. On the other hand, else the testing requirements will be inserted again in Stage 3 and the identification process will be resumed.

In Step 2 We are building a set of sequential programs. Given the experience acquired, our proposal is use search-based to optimize *Stage 3 – Application of properties* and, likewise, include human perception. The performance of identification algorithm will be measured based on the number of testing requirements identified and, consequently, an increase in the coverage rate of the test.

Since there are no properties for identifying infeasible testing requirements for concurrent programs, Step 3 will carry out an empirical study to identify properties for infeasible testing requirements in concurrent programs. Our research group has a set of concurrent programs that have been used in studies of automatic generation of test cases. Thus, we intend to empirically analyze the

<sup>1</sup><https://github.com/JoaoChoma/snowballing>

<sup>2</sup><https://github.com/JoaoChoma/empiricalstudy>



**Figure 1: Envisioned Approach to Infeasible Testing Requirements Identification.**

experimental results of the group and seek for properties that reveal infeasible synchronization sequences. In Step 4, we will extend the algorithm presented in Step 2 and automate the application of the properties identified in Step 3.

In Step 5, we will use search-based algorithms to optimize the algorithm of Step 4 to mitigate the infeasible synchronization sequences in concurrent programs. Initially we will use the genetic algorithm to search synchronization sequences that present properties of infeasible testing requirements. Later, it will be possible to compare the performance with other search algorithms.

Finally, in Step 6, will use tester's perception to ensure that all identified properties are classified, since only algorithm will not be able to make the decision in all cases. Step 6 will be applied in Steps 2 and 5.

Steps 4 and 5 will be evaluated with concurrent program source codes, already used in other studies in our research group, because these studies report difficulties related to existence of infeasible testing requirements. The performance of identification algorithm will be measured based on number of testing requirements identified and, consequently, an increase in coverage rate of the test.

## 4 EXPECTED CONTRIBUTIONS

We expect that our approach will be able to automatically identify infeasible test requirements and, thereby, minimize the incidence of the problem of non-executability, in sequential and concurrent programs. In addition, the use of search-based algorithms will optimize the process of identifying infeasible testing requirements in concurrent programs, which will minimize tester's effort to identify infeasible testing requirements manually. As a way of improvement the proposal, the use of human perception will adjust the decision-making process regarding the identification of infeasible testing requirements. As a future work, use of a machine learning algorithm to replace the tester in decision making with the aim of achieving 100% automation with high accuracy.

## ACKNOWLEDGMENTS

The authors acknowledge the São Paulo Research Funding, FAPESP, for the financial support under process no. 2018/25744-6.

## REFERENCES

- [1] N Anastopoulos and N Koziris. 2008. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–8.
- [2] B Barhoush and I Alsmadi. 2013. Infeasible Paths Detection Using Static Analysis. *Ijg.Acm.Org* II, Iii (2013).
- [3] R Bodík, R Gupta, and M L Soffa. 1997. Refining data flow information using infeasible paths. *ACM SIGSOFT Software Engineering Notes* 22, 6 (1997), 361–377.
- [4] P M S Bueno and M Jino. 2000. Identification of potentially infeasible program paths by monitoring the search for test data. *Proceedings ASE 2000: 15th IEEE International Conference on Automated Software Engineering* (2000), 209–218.
- [5] IS Chung, Hyeyon S Kim, H S Bae, Y R Kwon, and D G Lee. 1999. Testing of concurrent programs after specification changes. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 199–208.
- [6] Lori A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering* 3 (1976), 215–222.
- [7] S S Dahiya, J K Chhabra, and S Kumar. 2011. *PSO Based Pseudo Dynamic Method for Automated Test Case Generation Using Interpreter*. 147 pages. arXiv:9780201398298
- [8] M Delahaye, B Botella, and A Gotlieb. 2015. Infeasible path generalization in dynamic symbolic execution. *Information and Software Technology* 58 (2015), 403–418.
- [9] Carl Dionne, Marc Feeley, and Jocelyn Desbien. 1996. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *PDPTA*. 203–214.
- [10] P G Frankl. 1987. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. Ph.D. Dissertation. New York, NY, USA. AAI8801533.
- [11] D Gong, T Tian, and X Yao. 2012. Grouping target paths for evolutionary generation of test data in parallel. *Journal of Systems and Software* 85, 11 (2012), 2531–2540.
- [12] D. Hedley and M. A. Hennell. 1985. The causes and effects of infeasible paths in computer programs. (1985), 259–266.
- [13] I. Hermadi and M. A. Ahmed. 2003. Genetic Algorithm based Test Data Generator. *The 2003 Congress on Evolutionary Computation, 2003. CEC '03. May* (2003), 184.
- [14] I. Hermadi, C. Lokan, and R. Sarker. 2014. Dynamic stopping criteria for search-based test data generation for path testing. *Information and Software Technology* 56, 4 (2014), 395–407.
- [15] D Kundu, M Sarma, and D Samanta. 2015. A UML model-based approach to detect infeasible paths. *Journal of Systems and Software* 107 (2015), 71–92.
- [16] Y Lei and R Cz. 2005. A new algorithm for reachability testing of concurrent programs. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 10–pp.
- [17] N. Malevris, D. F. Yates, and A. Veevers. 1990. Predictive metric for likely feasibility of program paths. *Information and Software Technology* 32, 2 (1990), 115–118.
- [18] A W Marashdih and Z F Zaaba. 2018. Infeasible paths in static analysis: Problems and challenges. *AIP Conference Proceedings* 2016, September (2018).

- [19] P. McMinn. 2011. Search-based software testing: Past, present and future. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*. IEEE, 153–163.
- [20] G J Myers, C Sandler, and T Badgett. 2011. *The art of software testing*. John Wiley & Sons.
- [21] M N Ngo and H B K Tan. 2007. Detecting large number of infeasible paths through recognizing their patterns. *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07* (2007), 215.
- [22] M N Ngo and Hee B K Tan. 2008. Heuristics-based infeasible path detection for dynamic test data generation. *Information and Software Technology* 50, 7-8 (2008), 641–655.
- [23] P Pacheco. 2011. *An introduction to parallel programming*. Elsevier.
- [24] K Petersen, S Vakkalanka, and L Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.
- [25] G H.L. Pinto and S R Vergilio. 2010. A multi-objective genetic algorithm to test data generation. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI 1* (2010), 129–134.
- [26] SRS Souza, Silvia Regina Vergilio, PSL Souza, AS Simao, and Alexandre Ceolin Hausen. 2008. Structural testing criteria for message-passing parallel programs. *Concurrency and Computation: Practice and Experience* 20 (November 2008), 1893–1916. Issue 16.
- [27] K Tai et al. 1999. Automated test sequence generation using sequencing constraints for concurrent programs. In *Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on*. IEEE, 97–108.
- [28] H. Takagi. 2015. Interactive evolutionary computation for analyzing human characteristics. In *Emergent Trends in Robotics and Intelligent Systems*. Springer, 189–195.
- [29] S R Vergilio, J C Maldonado, and M Jino. 1992. Non-executable paths: Characterization, Prediction and Determination to Support Program Testing - In Portuguese.
- [30] S R Vergilio, J C Maldonado, and M Jino. 2006. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. *Journal of the Brazilian Computer Society* 12, 1 (2006), 73–88.
- [31] C Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) (EASE '14). ACM, New York, NY, USA, Article 38, 10 pages.
- [32] D Yates and N Malevris. 1989. Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Software Engineering Notes* 14, 8 (1989), 48–54.

# Author Index

- Abdessalem, Raja Ben ..... 88  
 Afzal, Wasif ..... 337  
 Alhanahnah, Mohannad ..... 272  
 Araujo Rodriguez, Luis Gustavo ..... 582
- Bagheri, Hamid ..... 272  
 Bartocci, Ezio ..... 569  
 Bell, Jonathan ..... 249  
 Bezzo, Nicola ..... 349  
 Briand, Lionel C. ..... 88  
 Bultan, Tevfik ..... 260  
 Busse, Frank ..... 63
- Cadar, Cristian ..... 63  
 Celik, Ahmet ..... 249  
 Černý, Pavol ..... 189  
 Cha, Sooyoung ..... 165  
 Chandra, Satish ..... 225  
 Chen, Bihuan ..... 376  
 Chen, Tao ..... 286  
 Chen, Yuqi ..... 14  
 Chen, Zhenyu ..... 177, 573  
 Chen, Zhong ..... 553  
 Choma Neto, João ..... 587  
 Choudhary, Rutvik ..... 211  
 Coley, Matthew ..... 249  
 Coppa, Emilio ..... 1  
 Cygan, Artur ..... 557
- D'Elia, Daniele Cono ..... 1  
 Deng, Xuan ..... 488  
 Dong, Jin Song ..... 440  
 Dou, Wensheng ..... 363, 528  
 Dutta, Saikat ..... 211  
 Dwyer, Matthew B. ..... 349
- Eichberg, Michael ..... 428  
 Elbaum, Sebastian ..... 349  
 Ernst, Michael D. ..... 298
- Fan, Gang ..... 463  
 Fang, Chunrong ..... 177, 516  
 Feist, Josselin ..... 557  
 Feng, Yang ..... 177, 573  
 Fioraldi, Andrea ..... 1  
 Fourtounis, George ..... 388  
 Fraser, Gordon ..... 440
- Gad, Ahmed ..... 452  
 Gallagher, John P. ..... 115  
 Gao, Jianbo ..... 553  
 Gao, Xinyu ..... 177  
 Gao, Yu ..... 363  
 Ghaleb, Asem ..... 415  
 Ghanbari, Ali ..... 75, 541  
 Gligoric, Milos ..... 249  
 Godefroid, Patrice ..... 312  
 Gopinath, Rahul ..... 27, 237  
 Grieco, Gustavo ..... 557  
 Groce, Alex ..... 557  
 Guan, Zhi ..... 553  
 Gullapalli, Vijay ..... 452  
 Guo, Chao ..... 545  
 Guo, Yue ..... 545
- Guo, Zichen ..... 549  
 Haller, Philipp ..... 428  
 Hao, Dan ..... 75  
 Hao, Rui ..... 545  
 Havrikov, Nikolas ..... 237  
 He, Tieke ..... 545, 549  
 He, Xiao ..... 502  
 Helm, Dominik ..... 428  
 Hildebrandt, Carl ..... 349  
 Huang, An ..... 153  
 Huang, Heqing ..... 38  
 Huang, Jeff ..... 516  
 Huang, Tianze ..... 363  
 Huang, Xin ..... 452  
 Huang, Yong ..... 573
- Jain, Aryaman ..... 211  
 Jiang, Muhui ..... 401  
 Jiang, Yanjie ..... 128  
 Jin, Jiahao ..... 128  
 Kadron, İsmet Burak ..... 260  
 Kampmann, Alexander ..... 237  
 Kölzer, Jan Thomas ..... 428  
 Kübler, Florian ..... 428
- Lam, Wing ..... 298  
 Lee, Dain ..... 165  
 Lee, Seokhyun ..... 165  
 Lehmann, Daniel ..... 312  
 Li, Hui ..... 363  
 Li, Ke ..... 286  
 Li, Qingshan ..... 553  
 Li, Xia ..... 75  
 Li, Xuandong ..... 153  
 Li, Xueliang ..... 115  
 Li, Yitong ..... 101  
 Li, Yue ..... 553  
 Li, Zhuoyang ..... 549  
 Lin, Yun ..... 440  
 Liu, Hui ..... 128  
 Liu, Jiawei ..... 549  
 Liu, Muyang ..... 286  
 Liu, Ting ..... 376, 440  
 Liu, Yang ..... 376, 401  
 Liu, Yepang ..... 115  
 Liu, Yi ..... 502  
 Liu, Zhibo ..... 475  
 Liu, Zixi ..... 516  
 Lou, Yiling ..... 75  
 Luo, Xiapu ..... 401  
 Lutellier, Thibaud ..... 101
- Ma, Shiqing ..... 565  
 Macêdo Batista, Daniel ..... 582  
 Machalica, Mateusz ..... 225  
 Manjunath, Niveditha ..... 569  
 Mariani, Leonardo ..... 141, 569  
 Mateis, Cristinel ..... 569  
 Mathis, Björn ..... 27  
 Meijer, Erik ..... 225  
 Men, Duo ..... 573  
 Mezini, Mira ..... 428
- Micucci, Daniela ..... 141  
 Milicevic, Aleksandar ..... 249  
 Misailovic, Sasa ..... 211  
 Mottadelli, Simone Paolo ..... 141  
 Murali, Vijayaraghavan ..... 225
- Nejati, Shiva ..... 88  
 Nićković, Dejan ..... 569  
 Nie, Pengyu ..... 249  
 Nowack, Martin ..... 63
- Oei, Reed ..... 298  
 Oh, Hakjoo ..... 165  
 Ostrand, Thomas J. ..... 337
- Pan, Minxue ..... 153  
 Pang, Lawrence ..... 101  
 Panichella, Annibale ..... 88  
 Pastore, Fabrizio ..... 569  
 Pattabiraman, Karthik ..... 415  
 Peng, Qianyang ..... 324  
 Pham, Hung Viet ..... 101  
 Polishchuk, Marina ..... 312  
 Poskitt, Christopher M. ..... 14  
 Pradel, Michael ..... 225
- Qian, Rebecca ..... 225  
 Qian, Ruixiang ..... 573
- Rall, Daniel ..... 452  
 Ren, Kui ..... 401  
 Riganelli, Oliviero ..... 141  
 Rinetzky, Noam ..... 51  
 Rosner, Nicolás ..... 260  
 Rota, Claudio ..... 141  
 Rubio-González, Cindy ..... 488
- Salvanesci, Guido ..... 428  
 Shao, Shuai ..... 565  
 Sharma, Arnab ..... 200  
 Shen, Mingzhu ..... 128  
 Shi, August ..... 211, 298, 324  
 Shi, Jia ..... 502  
 Shi, Qingkai ..... 38, 177, 463, 516, 573  
 Shi, Yangyang ..... 516  
 Smaragdakis, Yannis ..... 388  
 Song, Fu ..... 376  
 Song, Will ..... 557  
 Soremekun, Ezekiel O. ..... 237  
 Stevens, Clay ..... 272  
 Stifter, Thomas ..... 88  
 Strandberg, Per Erik ..... 337  
 Sullivan, Allison K. ..... 561  
 Sun, Jun ..... 14, 440  
 Sundmark, Daniel ..... 337
- Tan, Lin ..... 101  
 Tener, Greg ..... 452  
 Thompson, George ..... 561  
 Tizpaz-Niari, Saeid ..... 189  
 Trabish, David ..... 51  
 Triantafyllou, Leonidas ..... 388  
 Trivedi, Ashutosh ..... 189
- Vanover, Jackson ..... 488
- Wan, Jun ..... 177  
 Wang, Chengpeng ..... 463  
 Wang, Dong ..... 363  
 Wang, Guoxin ..... 153

Wang, Kaiyuan	452	Xu, Liang	528	Zhang, Charles	38, 463
Wang, Ruoyu	401	Xu, Lijie	363	Zhang, Fan	14
Wang, Shuai	475	Xu, Yifei	376	Zhang, Haotian	75
Wang, Wei	363	Xu, Zhengzi	376	Zhang, Lingming	75, 324
Wang, Xingwei	502	Xuan, Bohan	14	Zhang, Lu	75
Wehrheim, Heike	200	Xue, Feng	577	Zhang, Sai	298
Wei, Jun	363, 528	Yan, Wentian	553	Zhang, Tian	153
Wei, Moshi	101	Yang, Bo	528	Zhang, Yakun	528
Weyuker, Elaine J.	337	Yang, Yuming	115	Zhang, Yang	565
Wu, Kaishun	115	Yao, Peisen	38	Zhang, Zhekun	211
Wu, Rongxin	463	Ye, Dan	528	Zhangzhu, Peitian	549
Wu, Zhenhao	553	Yuan, Wei	545	Zhao, Yuan	573
Xiao, Xiao	463	Zeller, Andreas	27, 237	Zhong, Hua	363
Xie, Tao	298	Zhai, Juan	565	Zhou, Yajin	401
Xiu, Ziheng	440			Zhou, Zhiyong	528
				Zhu, Jiaxin	528