

A NEW COMEDY
FROM THE GUYS THAT CREATED
SHAUN OF THE DEAD

THEY'RE BAD BOYS.
THEY'RE DIE HARDS.
THEY'RE LETHAL WEAPONS.
THEY ARE...

HOT FUZZ

SIMON PEGG NICK FROST

ROGUE PICTURES PRESENTS IN ASSOCIATION WITH STUDIOCANAL A WORKING TITLE PRODUCTION IN ASSOCIATION WITH BIG TALK PRODUCTIONS
"HOT FUZZ" SIMON PEGG NICK FROST JIM GIRDLEBERT JESS NINA GOLD PROD BY DAVID ARNOLD EDITOR ANNE BARONCE
WRITTEN BY CHRIS DICKENS PRODUCED BY NICHOLAS ROWLAND DIRECTED BY JESS HALL MUSIC BY RONALDO VASCONCELOS COSTUME DESIGNER NATASHA WHARTON
EXECUTIVE PRODUCERS EDGAR WRIGHT & SIMON PEGG PRODUCED BY NICKY PARK TIM BEVAN ERIC FELLNER PRODUCED BY EDGAR WRIGHT



ROGUE PICTURES

www.jointhefuzz.com





**YOU CAN'T FIND
BUGS WITH SUPER
SIMPLE TECHNIQUES**

imgflip.com



**FUZZER GO
BRRRRRRRRRRR**

whoami

- PhD student @ ANU
- Team lead @ Defence Science and Technology Group
- Interests in fuzzing, (binary) program analysis



whoami

- PhD student @ ANU
- Team lead @ Defence Science and Technology Group
- Interests in **fuzzing**, (binary) program analysis



Outline

1. What is fuzzing?
2. Shades of fuzzers
 - Black, grey, white
3. Fuzzing research state-of-the-art
4. Future directions

What is fuzzing?

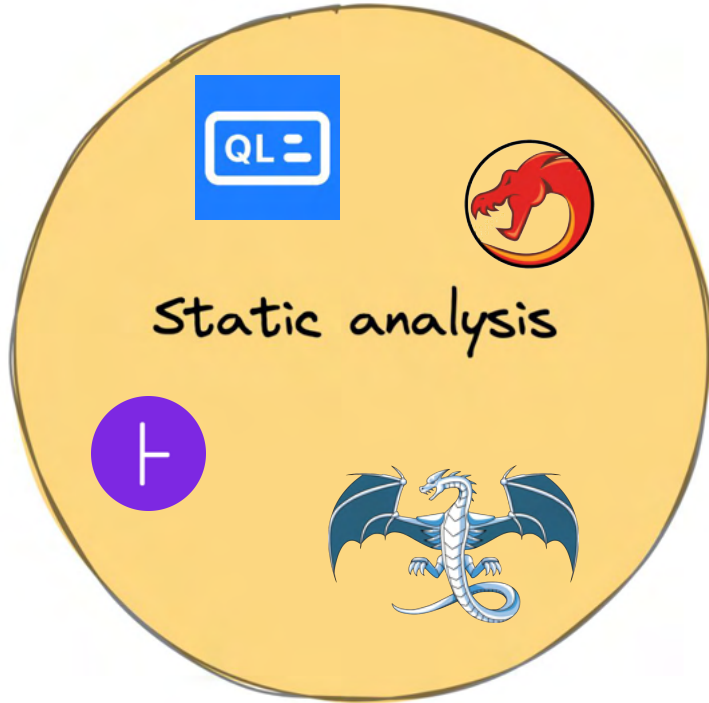


Static analysis

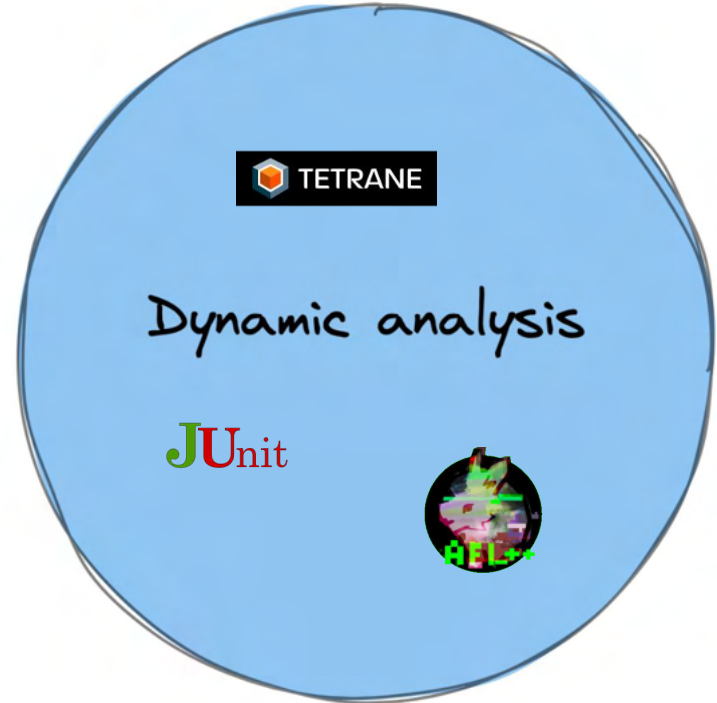
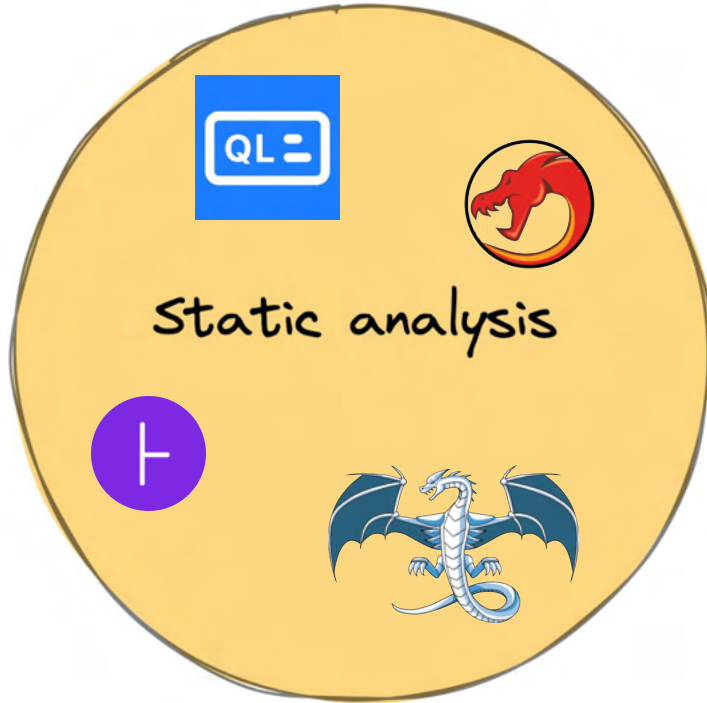


Dynamic analysis

What is fuzzing?



What is fuzzing?



What is fuzzing?

Pros

- No false positives
- Produces PoC
- Scalable



What is fuzzing?

Pros

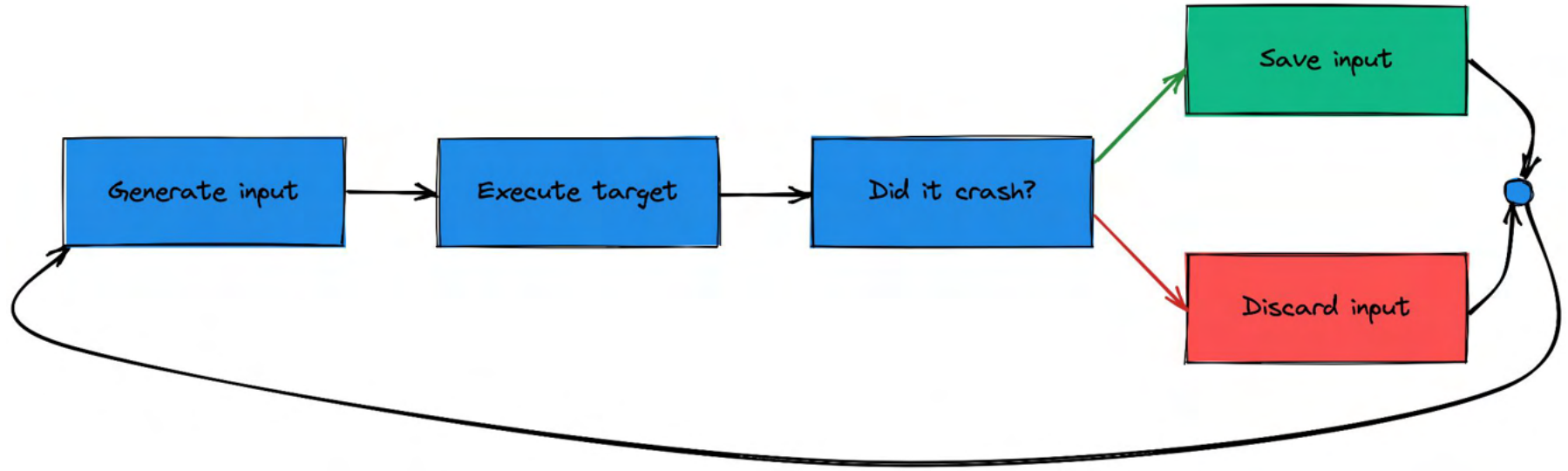
- No false positives
- Produces PoC
- Scalable

Cons

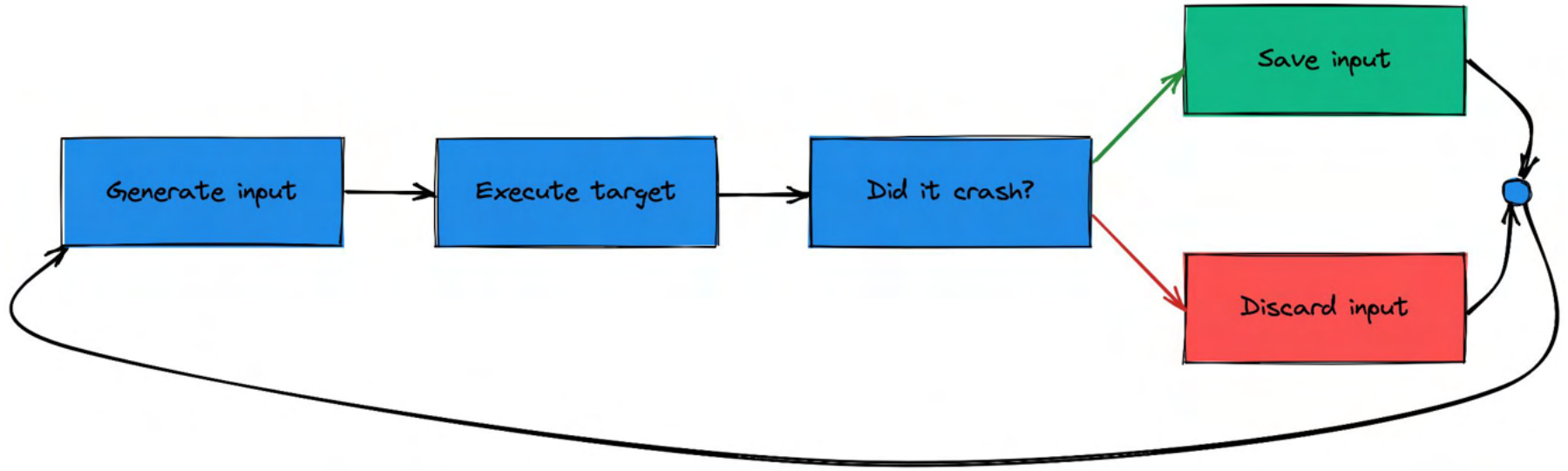
- Incomplete
- Requires buildable target
- Scalability



Our first fuzzer

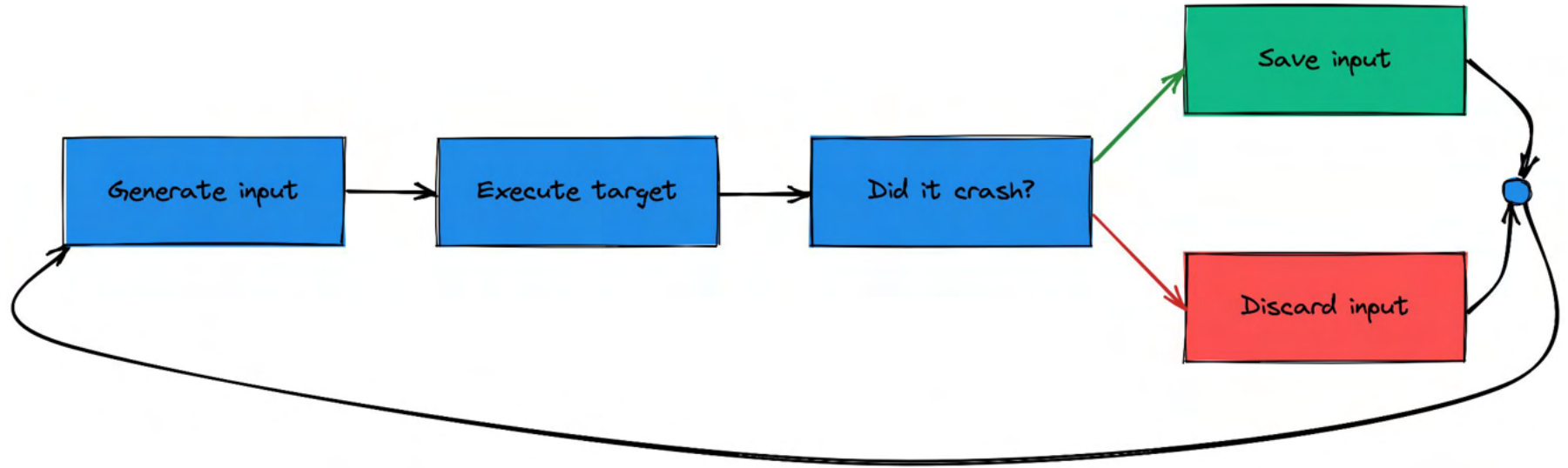


Our first fuzzer



How is this different to dynamic testing?

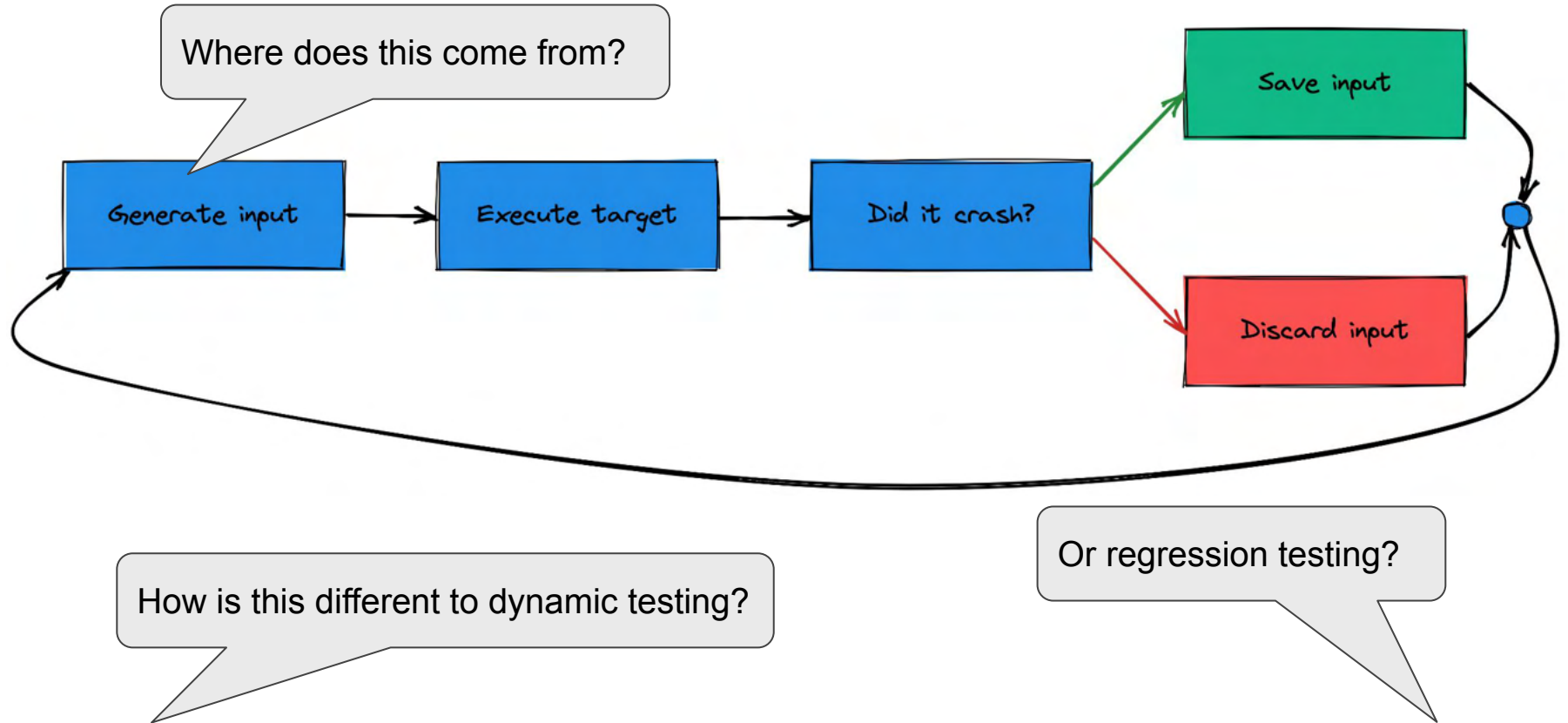
Our first fuzzer



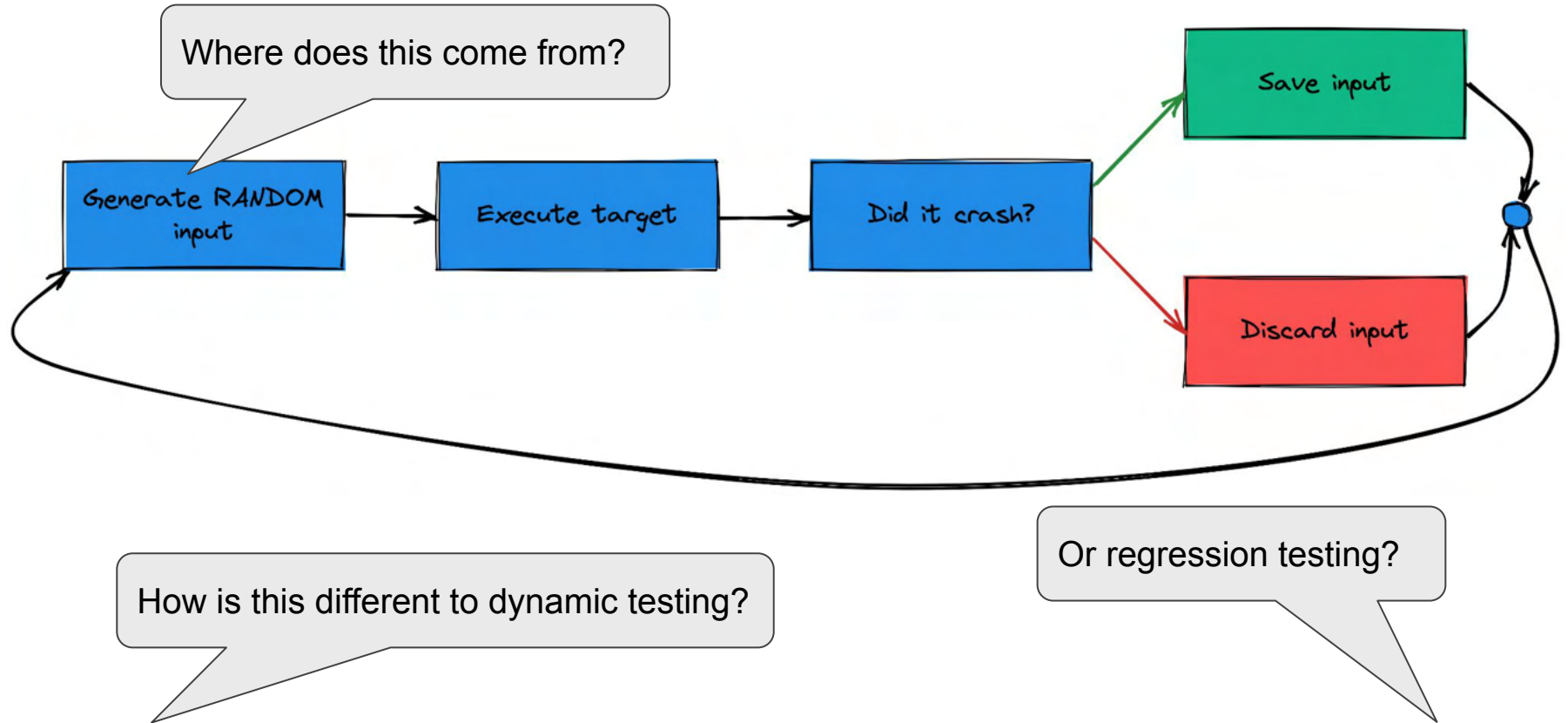
How is this different to dynamic testing?

Or regression testing?

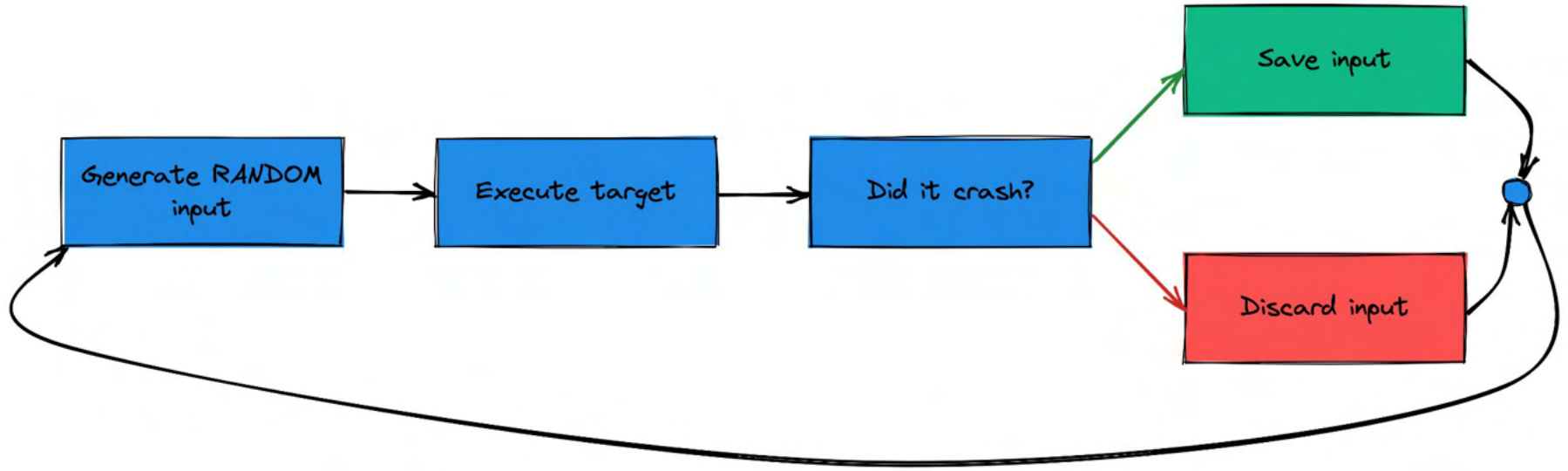
Our first fuzzer



Our first fuzzer



Our first fuzzer



A classic **generational blackbox** fuzzer

“An Empirical Study of the Reliability of Unix Utilities”

- Class project in 1988
“Advanced Operating Systems” course @ University Wisconsin

- Later published in 1990

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well tested and working correctly. To make a systematic statement about

Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities; (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings for types of strings that crash these programs were identified; and (4) the causes of the

to the Internet worm (the “ignifugus” bug) [2,5]. We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carefully typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to

An Empirical

the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing. On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to rue to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising, but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash (“core dump”); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the

program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix™, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A hypothesis of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedure, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashes or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons. First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes

edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.



While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and, simplify the generation of

Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of UNIX Utilities



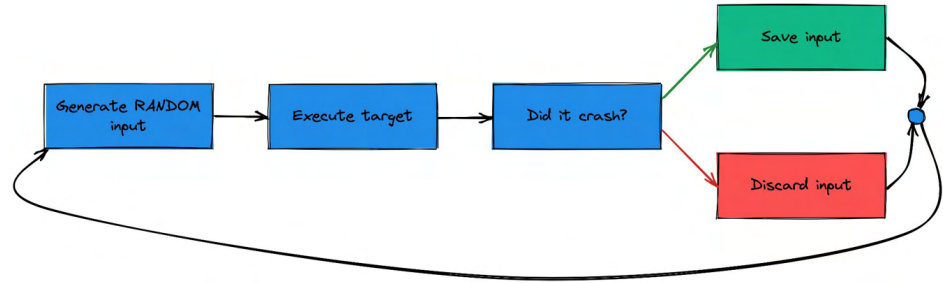
Blackbox fuzzers

- Radamsa
 - General-purpose file mutator
- Domato 
 - Google ProjectZero DOM fuzzer
- Peach 
 - Web API fuzzing (REST, SOAP, GraphQL)

Blackbox fuzzing

Pros

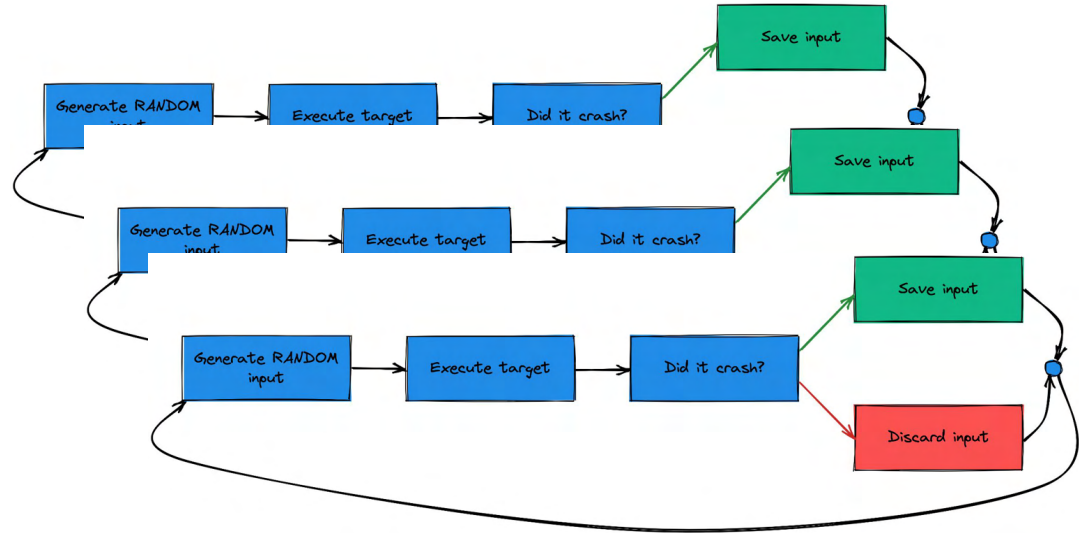
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Pros

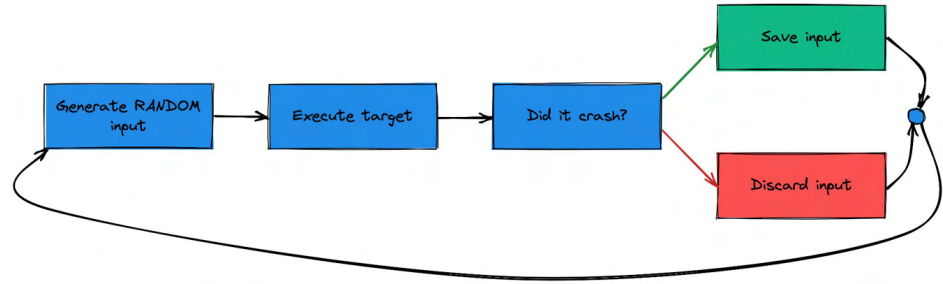
- Simple
- Fast
- Embarrassingly parallel



Blackbox fuzzing

Cons

- Generate mostly rubbish
- No notion of “progress”
- Only detect `SIGSEGV`

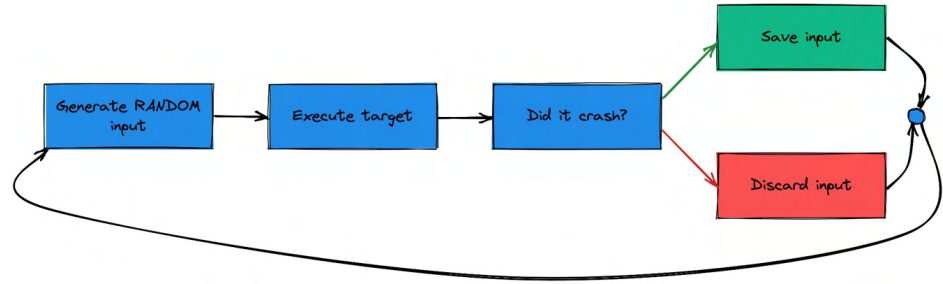


Can we do better?

Blackbox fuzzing

Cons

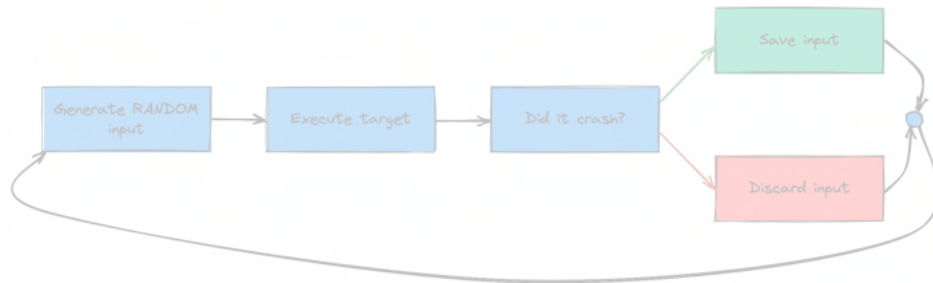
- Generate mostly rubbish
- No notion of “progress”
- Only detect `SIGSEGV`



Blackbox fuzzing

Cons

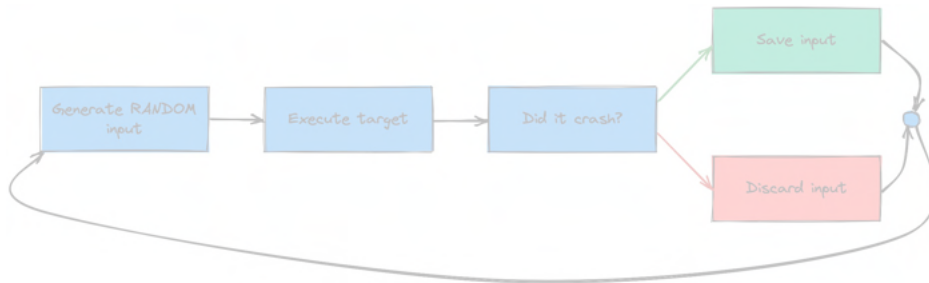
- Generate mostly rubbish
 - ~~Generate~~ **mutate**
- No notion of “progress”
- Only detect SIGSEGV



Blackbox fuzzing

Cons

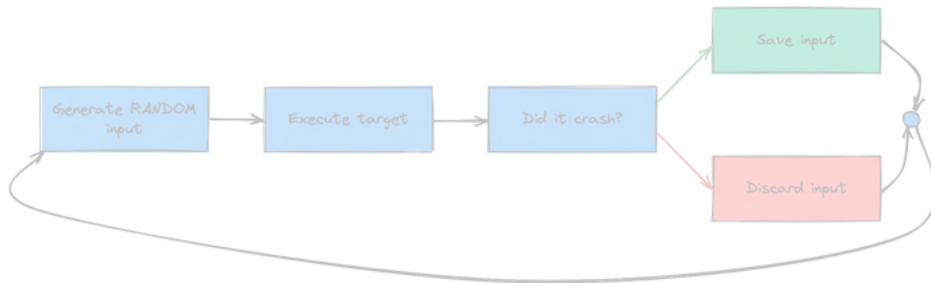
- Generate mostly rubbish
 - ~~Generate~~ mutate
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV



Blackbox fuzzing

Cons

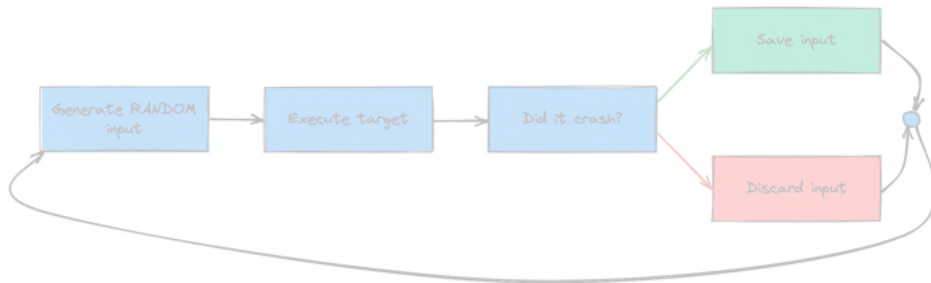
- Generate mostly rubbish
 - ~~Generate~~ **mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**



Blackbox fuzzing

Cons

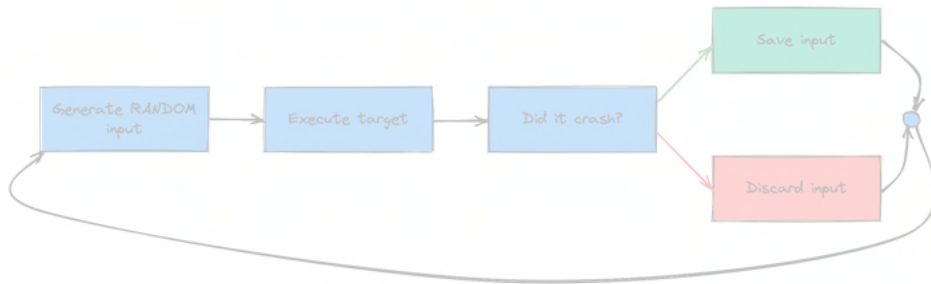
- Generate mostly rubbish
 - **Generate mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**



Blackbox fuzzing

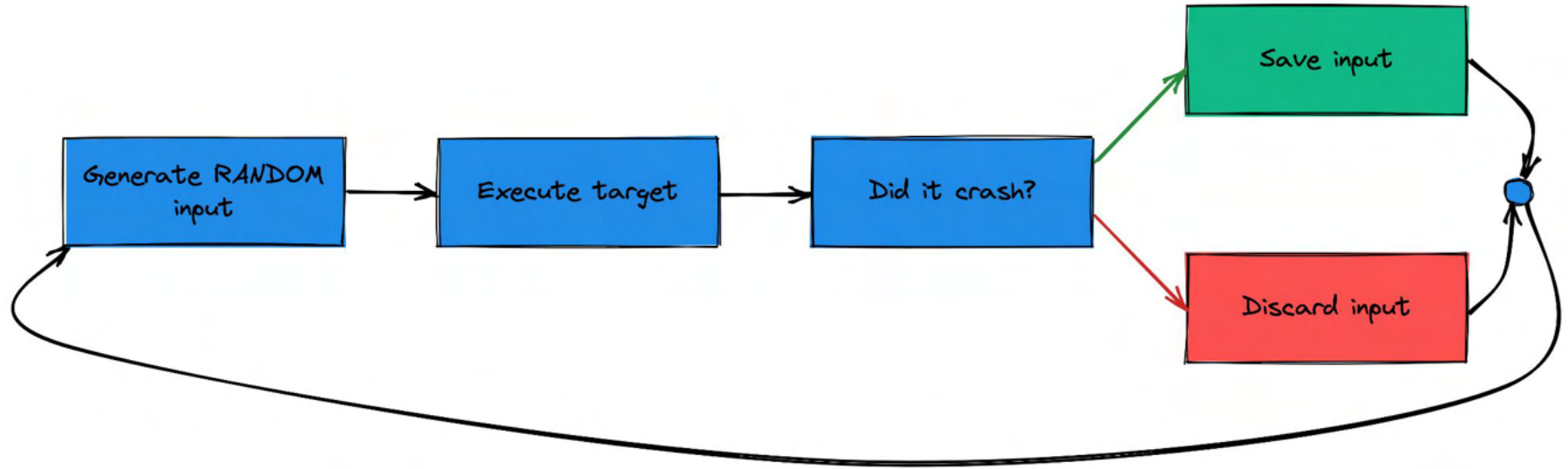
Cons

- Generate mostly rubbish
 - **Generate mutate**
- No notion of “progress”
 - Add a **feedback loop**
- Only detect SIGSEGV
 - Add a **sanitizer**

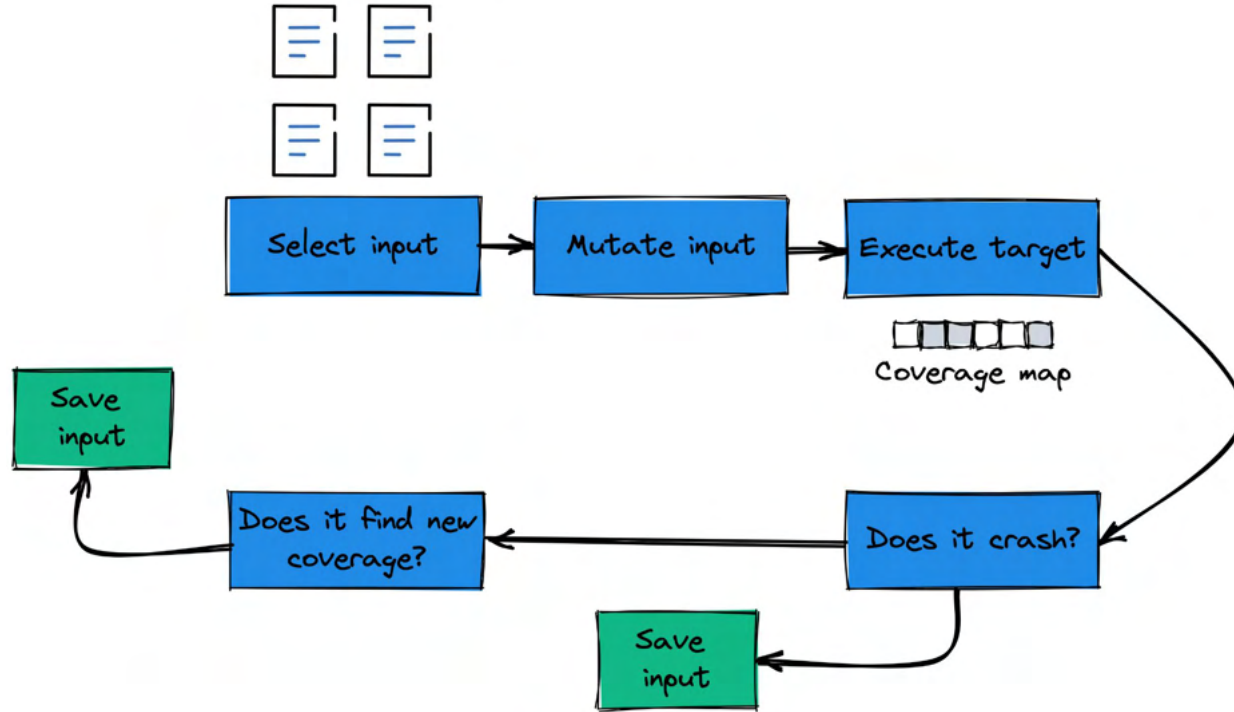


Mutational coverage-guided fuzzer
aka
greybox fuzzer

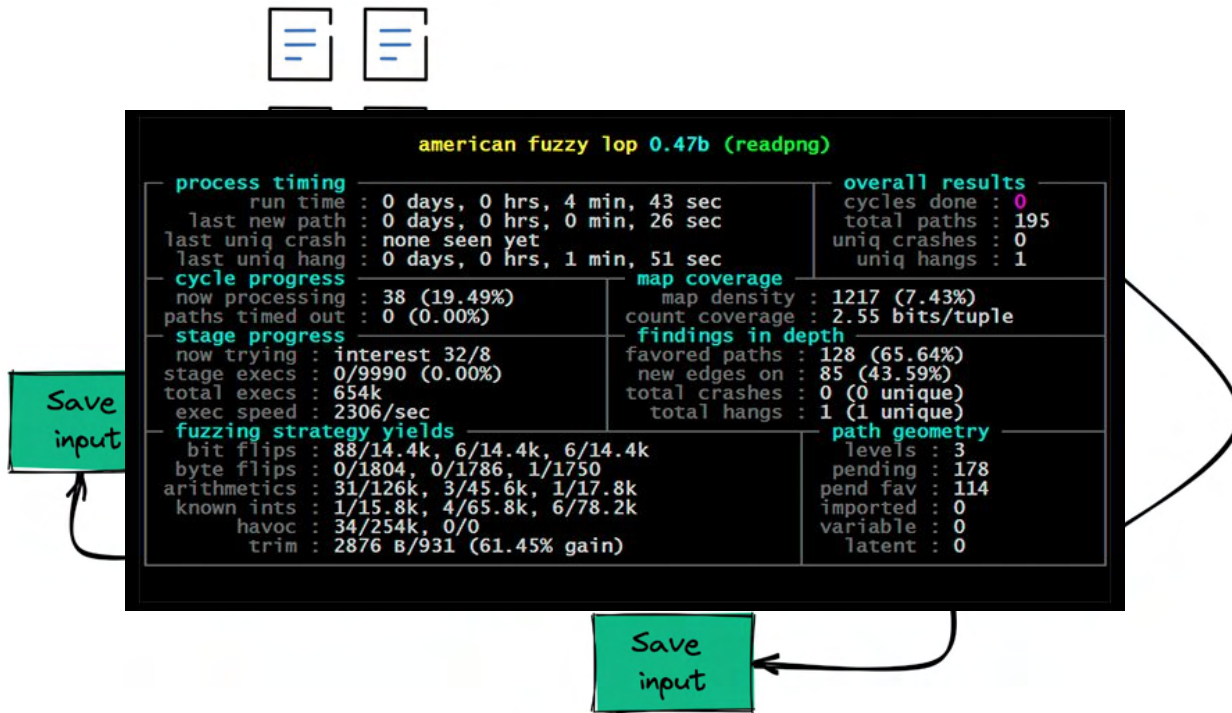
Blackbox fuzzing



Greybox fuzzing



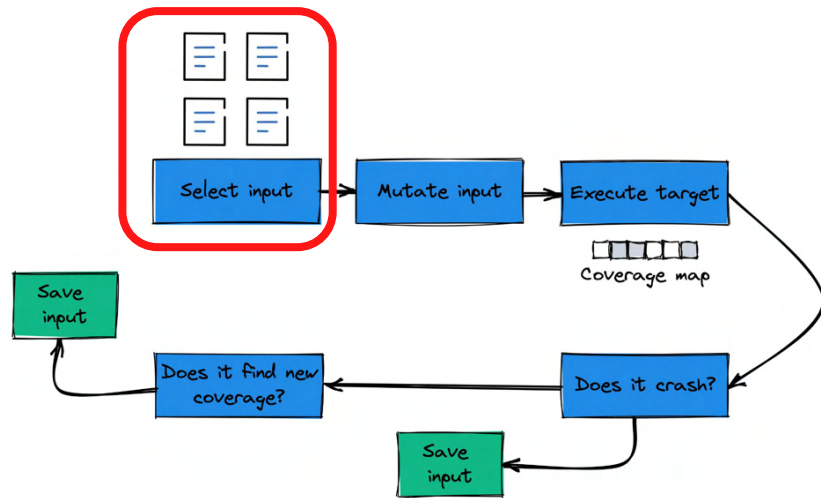
Greybox fuzzing



Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

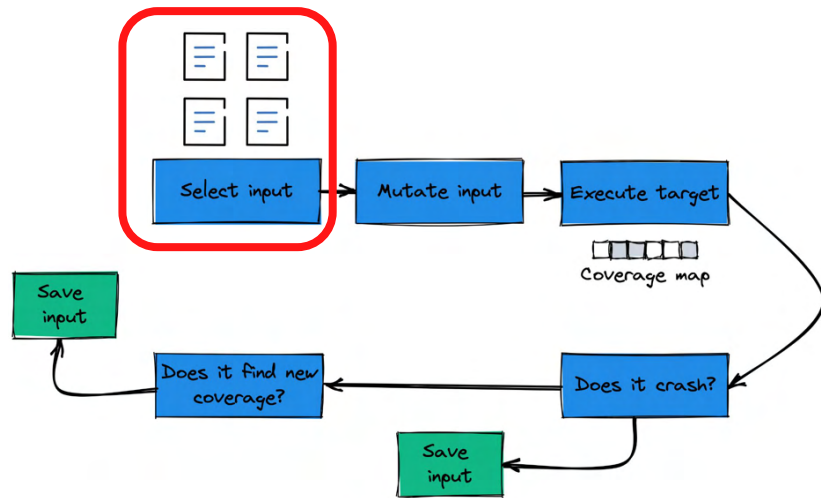


Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

Where do these initial inputs come from?



Seed selection

- In academic evaluations: “empty seed” common
- In practice: large corpora

Seed selection

- In academic evaluations: “empty seed” common
- In practice: large corpora

Which is better?

Seed selection

Optimizing Seed Selection for Fuzzing

Alexandre Rebert^{1,5} Sang Kil¹
alex@forallsecure.com sangkilc@cs

David Warren¹ G
dwarren@cert.org gg@c

¹ Carnegie Mellon University
⁵ Software l

Seed Selection for Successful Fuzzing

Adrian Herrera
ANU & DST
Australia

Hendra Gunadi
ANU
Australia

Shane Magrath
DST
Australia

Michael Norrish
CSIRO's Data61 & ANU
Australia

Mathias Payer
EPFL
Switzerland

Antony L. Hosking
ANU & CSIRO's Data61
Australia

Abstract

Randomly mutating well-formed program inputs, *fuzzing*, is a highly effective and widely used¹ to find bugs in software. Other than showing *fuzz* bugs, there has been little systematic effort in understanding the science of how to fuzz properly. In this, we focus on how to mathematically formulate and about one critical aspect in fuzzing: how best to pick files to maximize the total number of bugs found a fuzz campaign. We design and evaluate six algorithms using over 650 CPU days on Amazon Compute Cloud (EC2) to provide ground truth. Overall, we find 240 bugs in 8 applications and the choice of algorithm can greatly increase the # of bugs found. We also show that current seed strategies as found in Peach may fare no better than seeds at random. We make our data set an publicly available.

1 Introduction

Software bugs are expensive. A single software is enough to take down spacecrafts [2], make centrifuges spin out of control [17], or recall 1000 faulty cars resulting in billions of dollars in damage. In 2012, the software security market was estimated at \$19.2 billion [12], and recent forecasts predict an increase in the future despite a sequestering economy. The need for finding and fixing bugs in software they are exploited by attackers has led to the development of sophisticated automatic software testing tools. Fuzzing is a popular and effective choice for bugs in applications. For example, fuzzing is a part of the overall quality checking process employed by Adobe [28], Microsoft [14], and Google [27], as

ABSTRACT

Mutation-based greybox fuzzing—unquestionably the most widely-used fuzzing technique—relies on a set of non-crashing seed inputs (a corpus) to bootstrap the bug-finding process. When evaluating a fuzzer, common approaches for constructing this corpus include: (i) using an empty file, (ii) using a single seed representative of the target's input format, or (iii) collecting a large number of seeds (e.g., by crawling the Internet). Little thought is given to how this seed choice affects the fuzzing process, and there is no consensus on which approach is best (or even if a best approach exists).

To address this gap in knowledge, we systematically investigate and evaluate how seed selection affects a fuzzer's ability to find bugs in *real-world* software. This includes a systematic review of seed selection practices used in both evaluation and deployment contexts, and a large-scale empirical evaluation (over 650 CPU-years) of six seed selection approaches. These six seed selection approaches include three *corpus minimization* techniques (which select the smallest subset of seeds that trigger the same range of instrumentation data points as a full corpus).

Our results demonstrate that fuzzing outcomes vary significantly depending on the initial seeds used to bootstrap the fuzzer, with minimized corpora outperforming singleton, empty, and large (in the order of thousands of files) seed sets. Consequently, we encourage seed selection to be foremost in mind when evaluating/deploying fuzzers, and recommend that (a) seed choice be carefully considered and explicitly documented, and (b) never to evaluate fuzzers with only a single seed.

CCS CONCEPTS

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

KEYWORDS

fuzzing, corpus minimization, software testing

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 that may not be registered with 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 permissions from permissions@acm.org.
DST, 21, July 11–17, 2021, Virtual, Denmark.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8459-5/21/07.
<https://doi.org/10.1145/3601573.3667795>

ACM Reference Format:

Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (DSTA '21)*, July 11–17, 2021, Virtual, Denmark. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3601573.3667795>

1 INTRODUCTION

Fuzzing is a dynamic analysis technique for finding bugs and vulnerabilities in software, triggering crashes in a target program by subjecting it to a large number of (possibly malformed) inputs. *Mutation-based* fuzzing typically uses an initial set of valid seed inputs from which to generate new seeds by random mutation. Due to their simplicity and ease-of-use, mutation-based greybox fuzzers such as AFL [74], honggfuzz [64], and libFuzzer [61] are widely deployed, and have been highly successful in uncovering thousands of bugs across a large number of popular programs [6, 16]. This success has prompted much research into improving various aspects of the fuzzing process, including mutation strategies [19, 42], *energy assignment policies* [15, 25], and *path exploration algorithms* [14, 73]. However, while researchers often note the importance of high-quality input seeds and their impact on fuzzer performance [37, 56, 58, 67], few studies address the problem of *optimal design and construction of corpora for mutation-based fuzzers* [56, 58], and none assess the precise impact of these corpora in coverage-guided mutation-based greybox fuzzing.

Intuitively, the collection of seeds that form the initial corpus should generate a broad range of observable behaviors in the target. Similarly, candidate seeds that are behaviorally similar to one another should be represented in the corpus by a single seed. Finally, both the total size of the corpus and the size of individual seeds should be minimized. This is because previous work has demonstrated the impact that file system contention has on industrial-scale fuzzing. In particular, Xu et al. [71] showed that the overhead from opening/closing test-cases and synchronization between workers each introduced a 2x overhead. Time spent opening/closing test-cases and synchronization is time diverted from mutating inputs and expanding code coverage. Minimizing the total corpus size and the size of individual test-cases reduces this wastage and enables time to be (better) spent on finding bugs.

Under these assumptions, simply gathering as many input files as possible is not a reasonable approach for constructing a fuzzing corpus. Conversely, these assumptions also suggest that beginning with the “empty corpus” (e.g., consisting of one zero-length file) may be less than ideal. And yet, as we survey here, the majority of published research uses either (a) the “singleton corpus” (e.g., a single seed representative of the target program's input format),

- Empty = easy to compare fuzzers
 - Only good for finding shallow bugs

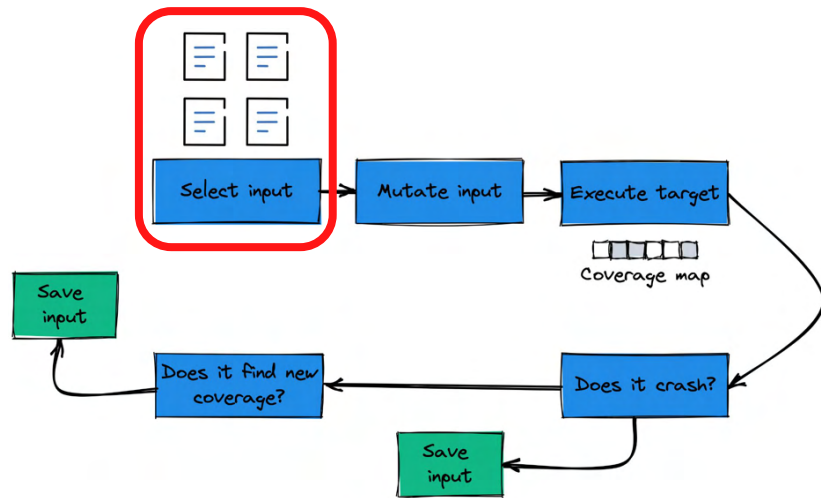
- Too large corpus = slow fuzzer

- Sweet spot: Use a corpus minimizer
 - Doesn't matter which one

Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data



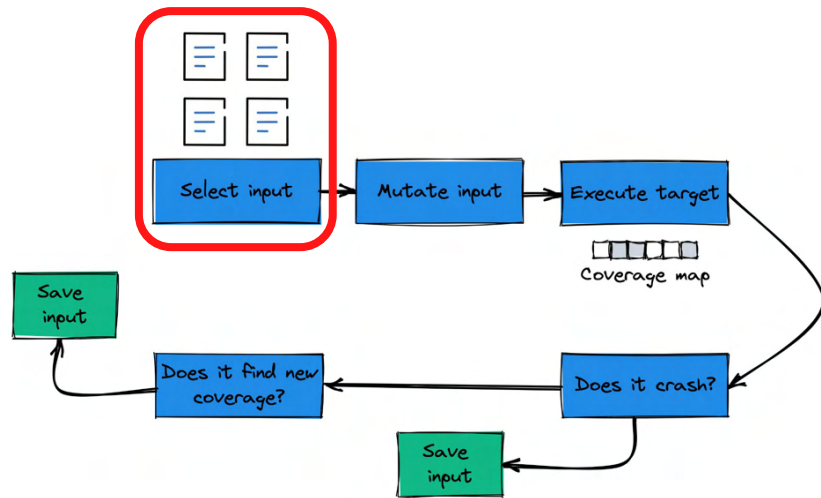
Greybox fuzzing

Select input

- Rather than generating random data, mutate existing data

How long do we focus on a seed?

How do we select this seed?



Power scheduling

- Power schedule = amount of energy assigned to an input

- Decrease energy each execution
- When energy = 0, change inputs

- Examples

- Markov chain
- Multi-arm bandit
- Machine learning
- Heuristics

Coverage-Based Greybox Fuzzing as Markov Chain

Marcel Böhme[✉], Van-Thuan Pham[✉], and Abhik Roychoudhury

Abstract—Coverage-based Greybox Fuzzing (CGF) generated by slightly mutating a seed input. If the seed is discarded, we observe that most tests exercise more paths with the same number of tests by gras CGF using a Markov chain model which specifies exercises path i . Each state (i.e., seed) has an energy that CGF is considerably more efficient if energy is monotonically every time that seed is chosen. Ens extending AFL to 24 hours, AFLFast exposes 3.9 unreported CVEs 7x faster than AFL, AFLFast per AFLFast to the symbolic executor Klee. In terms of same subject programs that were discussed in the Klee while a combination of both tools achieves be

Index Terms—Vulnerability detection, fuzzing, pm

1 INTRODUCTION

RECENTLY, there has been a controversial efficiency of symbolic execution-based fuzzers versus more lightweight greybox fuzzers. Symbolic execution is a systematic effort to st behaviors and thus considerably more effective most vulnerabilities were exposed by parts weight fuzzers that do not leverage any program. It turns out that even the most effective ted efficient than blackbox fuzzing if the time spea test case takes too long [4]. Symbolic execution effective because each new test exercises a diff the program. However, this effectiveness comes of spending significant time doing program analysis and solving. Blackbox fuzzing, on the other not require any program analysis and generate orders of magnitude more tests in the same time. Coverage-based Greybox Fuzzing (CGF) is a make fuzzing more effective at path exploration sacrificing time for program analysis and generate (binary) instrumentation to determine a unique for the path that is exercised by an input. Newly created by slightly mutating the provided seed also call the new tests as fuzz. If some fuzz ex

• The authors are with the Department of Computer Science, National University of Singapore, Singapore. E-mail: {bohme, thuan, abhik}@nus.edu.sg. Manuscript received 11 Aug. 2017; revised 4 Dec. 2017; accepted 20 Dec. 2017; date of publication 20 Dec. 2017; date of current version 22 Feb. 2018. Corresponding author: Marcel Böhme. Recommended for acceptance by X. Zhang. For information on obtaining reprints of this article, please write to the IEEE Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Copyright © 2018 IEEE. All rights reserved. DOI: 10.1109/TCSE.2017.2780441

EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit

Tai Yue, Pengfei Wang, Yong Tang*, Enze Wang, Bo Yu, Kai Lu, Xu Zhou

{yuetai17, p.

CEREBRO: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection

Yuekang Li University of Science and Technology of China Nanyang Technological University Singapore	Yinxing Xue* University of Science and Technology of China Nanyang Technological University Singapore	Hongxu Chen Nanyang Technological University Singapore
Xiuheng Wu Nanyang Technological University Singapore	Cen Zhang Nanyang Technological University Singapore	Xiaofei Xie Nanyang Technological University Singapore
Haijun Wang Nanyang Technological University Singapore	Yang Liu Nanyang Technological University Singapore Zhejiang Sci-Tech University China	

ABSTRACT

Existing greybox fuzzers mainly utilize program coverage as the goal to guide the fuzzing process. To maximize their outputs, coverage-based greybox fuzzers need to evaluate the quality of seeds properly, which involves making two decisions: 1) which is the most promising seed to fuzz next (seed prioritization), and 2) how many efforts should be made to the current seed (power scheduling). In this paper, we present our fuzzer, CEREBRO, to address the above challenges. For the seed prioritization problem, we propose an online multi-objective based algorithm to balance various metrics such as code complexity, coverage, execution time, etc. To address the power scheduling problem, we introduce the concept of input potential to measure the complexity of uncovered code and propose a cost-effective algorithm to update it dynamically. Unlike previous approaches where the fuzzer evaluates an input solely based on the execution traces that it has covered, CEREBRO is able to foresee the benefits of fuzzing the input by adaptively evaluating its input potential. We perform a thorough evaluation of CEREBRO on 8 different real-world programs. The experiments show that CEREBRO can find more vulnerabilities and achieve better coverage than state-of-the-art fuzzers such as AFL and AFLFast.

1 Introduction

Fuzzing is an automated software input and effective for detecting which was first devised by B. Since then, fuzzing has been of the most effective technique Fuzzing (CGF) has attracted se

CCS CONCEPTS

• Security and privacy → Vulnerability scanners.

*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 retained. Copying to reproduce this document requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. ESEC/FSE '18, August 26–30, 2018, Tallinn, Estonia. © 2018 Association for Computing Machinery. ACM ISBN 978-1-4503-5727-8/18/06...\$15.00. <https://doi.org/10.1145/318066.318075>

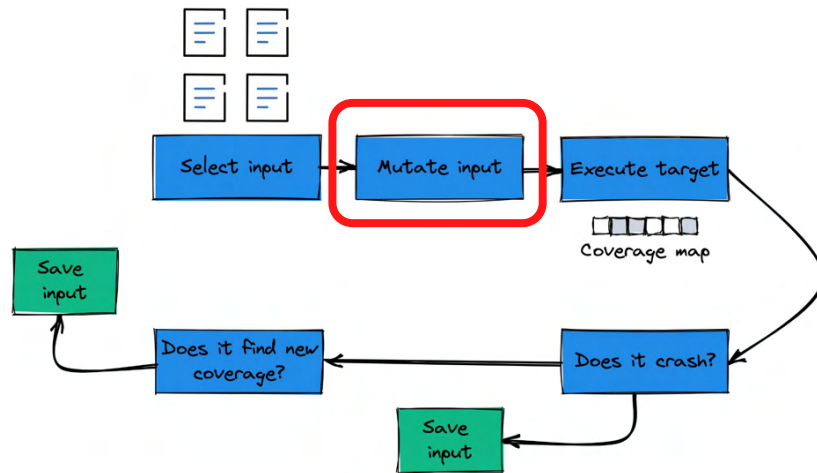


*In this paper, we denote all the files fed to the PUT by fuzzers as inputs, and only those inputs kept by fuzzers for subsequent mutations as seeds.

Greybox fuzzing

Mutate input

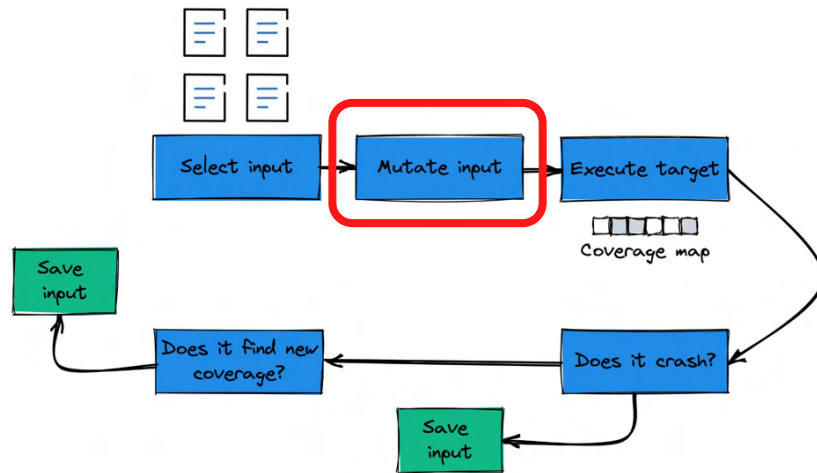
- Mutate enough to explore “interesting” states
- Don’t mutate too much, or we’ll just error out



Greybox fuzzing

Mutate input

- Mutate enough to explore “interesting” states
- Don’t mutate too much, or we’ll just error out



Where and how do we mutate?

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice

Structure aware

- Keyword substitution, grammar-based

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice
- Fast
- Simple to implement
- Destroys structure

Structure aware

- Keyword substitution, grammar-based
- Explore “deeper” code
- Require *a priori* knowledge

Mutations

Structure agnostic

- Bit flip, byte/word/... substitution, repetition, splice
- Fast
- Simple to implement
- Destroys structure

Structure aware

- Keyword substitution, **grammar-based**
- Explore “deeper” code
- Require *a priori* knowledge

Grammar-based fuzzing

- Many targets (e.g., JavaScript interpreter) accept input described by a **context-free grammar (CFG)**

- Highly structured
- Blind mutation will destroy structure

- Leverage CFG in mutation

- “Lift” input to parse tree
- Mutate parse tree(s)
- Lower parse tree back to file

Co
Ru
come

Technis
patrick.ja

Abstract—Fu
identifying bugs
that require big
fuzzed, many fu
interpreters etc
are passed, the
fuzzers from ex
interesting — c
execution of diff
many mutations
making small c
execution of erro
fuzzers are able
Free Grammars,
of fuzzing engi
mutational fuzz
grammar fuzzers
do not know wh

In this paper
fuzz programs th
the use of gram
This allows us t
to increase the
syntactically and
d-concept fuzzer
ChakraCore (th
murely, and Lau
the targets. Sever
one in Linux, 1
2600 USD and 6
combining code
significantly only
an order of mag
factor of two wh

Software co
life. Hence, the

Network and Distri
24-27 February 201
1839-1-891562-55.
https://doi.org/10
www.indus-symposi

This work is licensed under a Creative Commons
Attribution License.
© 2021 Copyright held by the owner/authors.
ACM ISBN 978-1-4503-3460-1
https://doi.org/10.1145/3460119.3460114

NAUTILUS:
Fishing for Deep Bugs with Grammars

Gramatron: Effective Grammar-Aware Fuzzing

Prashast Srivas
Purdue Univers
United States of A

GRIMOIRE: Synthesizing Structure while Fuzzing

Tim Blazytko, Cornelius Aschermann, Moritz Schlögl, Ali Abbasi,
Sergej Schumilo, Simon Wörner and Thorsten Holz

Ruhr-Universität Bochum, Germany

Abstract

In the past few years, fuzzing has received significant attention from the research community. However, most of this attention was directed towards programs without a dedicated parsing stage. In such cases, fuzzers which leverage the input structure of a program can achieve a significantly higher code coverage compared to traditional fuzzing approaches. This advancement in coverage is achieved by applying large-scale mutations in the application's input space. However, this improvement comes at the cost of requiring expert domain knowledge, as these fuzzers depend on structure input specifications (e.g., grammars). Grammar inference, a technique which can automatically generate such grammars for a given program, can be used to address this shortcoming. Such techniques usually infer a program's grammar in a pre-processing step and can miss important structures that are uncovered only later during normal fuzzing.

In this paper, we present the design and implementation of GRIMOIRE, a fully automated coverage-guided fuzzer which works without any form of human interaction or pre-configuration; yet, it is still able to efficiently test programs that expect highly structured inputs. We achieve this by performing large-scale mutations in the program input space using grammar-like combinations to synthesize new highly structured inputs without any pre-processing step. Our evaluation shows that GRIMOIRE outperforms other coverage-guided fuzzers when fuzzing programs with highly structured inputs. Furthermore, it improves upon existing grammar-based coverage-guided fuzzers. Using GRIMOIRE, we identified 19 distinct memory corruption bugs in real-world programs and obtained 11 new CVEs.

KEYWORDS

Fuzzing, grammar-aware, dynamic software

ACM Reference Format:
Prashast Srivas and Moritz Schlögl. 2021. Grammar-Aware Fuzzing. In *Proceedings of the 2021 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2021)*. ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3460119.3460114

1 Introduction

As the amount of software impacting the (digital) life of nearly every citizen grows, effective and efficient testing mechanisms for software become increasingly important. The publication of the fuzzing framework AFL [65] and its success at uncovering a huge number of bugs in highly relevant

software has spawned a large body of research on effective feedback-based fuzzing. AFL and its derivatives have largely conquered automated, dynamic software testing and are used to uncover new security issues and bugs every day. However, while great progress has been achieved in the field of fuzzing, many hard cases still require manual user interaction to generate satisfying test coverage. To make fuzzing available to more programmers and thus scale it to more and more target programs, the amount of expert knowledge that is required to effectively fuzz should be reduced to a minimum. Therefore, it is an important goal for fuzzing research to develop fuzzing techniques that require less user interaction and, in particular, less domain knowledge to enable more automated software testing.

Structured Input Languages. One common challenge for current fuzzing techniques are programs which process highly structured input languages such as interpreters, compilers, text-based network protocols or markup languages. Typically, such inputs are consumed by the program in two stages: parsing and semantic analysis. If parsing of the input fails, deeper parts of the target program—containing the actual application logic—fail to execute; hence, bugs hidden “deep” in the code cannot be reached. Even advanced feedback fuzzers—such as AFL—are typically unable to produce diverse sets of syntactically valid inputs. This leads to an imbalance, as these programs are part of the most relevant attack surface in practice, yet are currently unable to be fuzzed effectively. A prominent example are browsers, as they parse a multitude of highly-structured inputs, ranging from XML or CSS to JavaScript and SQL queries.

Previous approaches to address this problem are typically based on manually provided grammars or seed corpora [2, 14, 45, 52]. On the downside, such methods require human experts to (often manually) specify the grammar or suitable seed corpora, which becomes next to impossible for applications with undocumented or proprietary input specifications. An orthogonal line of work tries to utilize advanced program analysis techniques to automatically infer grammars

Grammar-based fuzzing

Pros

- Reach “deeper” code
- Can be used without coverage

Cons

- Require a priori knowledge of input format

Grammar-based fuzzing

Pros

- Reach “deeper” code
- Can be used without coverage

Cons

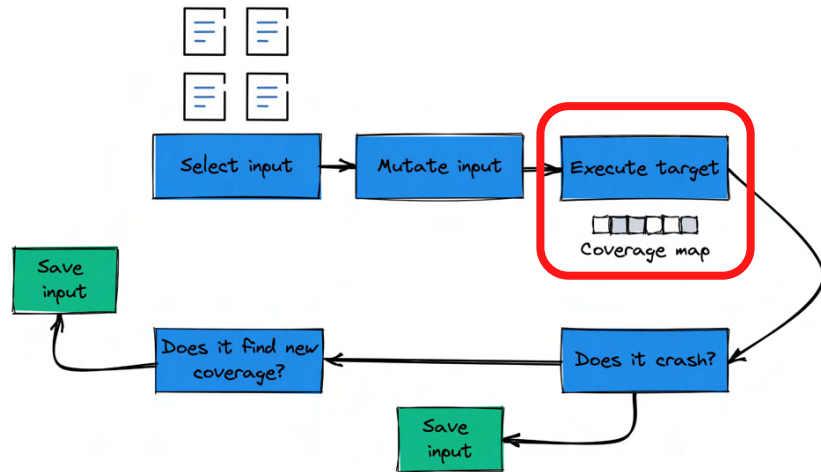
- Require a priori knowledge of input format

Some fuzzers try to “learn” this input format

Greybox fuzzing

Execute target

- Measure fuzzer “progress”
- Progress = code coverage



Coverage map

- Edge coverage is standard
- What if `# edges > sizeof(cov_map)`?
 - Must approximate
 - AFL uses a (lossy) hash function
- What if source is not available?
 - Use binary instrumentation (e.g., Intel PIN, DynamoRIO)

Edge coverage is a (relatively) poor approximation of a program's **state space**

- Context-sensitive edge
- Path
- Data flow

Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University
[†]{jwang131,wsong008}@ucr.edu, {heng.csong}@cs.ucr.edu [‡]yd375@cornell.edu

Registered Report: DATAFLOW

Towards a Data-Flow-Guided Fuzzer

Antony L. Hosking
ANU
antony.hosking@anu.edu.au

Coverage-guided g is the most common technique. It uses a coverage metric, which decides the essential parameter of the results. While there are many different coverage metrics, it is known about how they affect the fuzzing results. It is unclear whether coverage-guided is superior to all the other techniques.

We present **DATAFLOW**, a greybox fuzzer driven by lightweight data-flow profiling. Whereas control-flow edges represent the order of operations in a program, data-flow edges capture the dependencies between operations that produce data values and the operations that consume them: indeed, there may be no control dependence between those operations. As such, data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we establish a framework for reasoning about data-flow coverage, allowing the computational cost of exploration to be balanced with precision.

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for uncovering bugs in most targets we fuzzed (72 % of them). However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

1 Introduction

Greybox fuzzing is a technique that has been widely used by companies such as Google and Microsoft (e.g., Trail of Bits' American Fuzzoparty Challenge (CGC), grm0r1's Greybox Fuzzing Challenge). It is more effective than blackbox fuzzing and symbolic execution and

USENIX Association

program analyses to model program and input structure, and continuously gather dynamic information about the target.

Leveraging dynamic information drives fuzzer efficiency. For example, *coverage-guided greybox fuzzers*—perhaps the most widely-used class of fuzzer—track code paths executed by the target.¹ This allows the fuzzer to focus its mutations or inputs reaching new code. Intuitively, a fuzzer cannot find bugs in code never executed, so maximizing the amount of code executed should maximize the number of bugs found. Code coverage serves as an approximation of program behavior, and expanding code coverage implies exploring program behaviors.

Coverage-guided greybox fuzzers are now pervasive. Their success [2] can be attributed to one fuzzer in particular: American Fuzzy Lop (AFL) [3]. AFL is a greybox fuzzer that uses lightweight instrumentation to track edges covered in the target's control-flow graph (CFG). A large body of research has built on AFL [4–12]. While improvements have been made, most fuzzers still default to edge coverage as an approximation of program behavior. *Is this the best we can do?*

In some targets, control flow offers only a coarse-grained approximation of program behavior. This includes targets whose control structure is decoupled from its semantics (e.g., LR parsers generated by *yacc*) [13]. Such targets require *data-flow* coverage [13–17]. Whereas control flow focuses on the order of operations in a program (i.e., branch and loop structures), data flow instead focuses on how variables (i.e., data) are defined and used [14]; indeed, there may be no control dependence between variable definition and use sites (see §III for details).

In fuzzing, data flow typically takes the form of *dynamic taint analysis* (DTA). Here, the target's input data is *tainted* at its definition site and tracked as it is accessed and used at runtime. Unfortunately, accurate DTA is difficult to achieve and expensive to compute (e.g., prior work has found DTA is expensive [18, 19], and its accuracy highly variable across implementations [18, 20]). Moreover, several real-world programs fail to compile under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [3], libFuzzer [21], and honggfuzz [22]) eschew DTA in favor of higher fuzzing throughput.

While lightweight alternatives to DTA exist (e.g., REDQUEEN [23], GREYONE [19]), the full potential of control- vs. data-flow based fuzzer coverage metrics have not yet been thoroughly explored. To support this exploration, we

¹Miller et al.'s original fuzzer [1] is now known as a *blackbox* fuzzer because it has no knowledge of the target's internals.

International Fuzzing Workshop (FUZZING) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-0
<https://dx.doi.org/10.14722/fuzzing.2022.23001>
www.ndss-symposium.org

Coverage map

Edge coverage is a (relatively) poor approximation of a program's **state space**

Alternatives:

- Context-sensitive edge
- Path
- Data flow

Accuracy vs performance

Fuzzing with Data Dependency Information

Alessandro Mantovani
EURECOM
mantovan@eurecom.fr

Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing

Jinghan Wang[†], Yue Duan[‡], Wei Song[†], Heng Yin[†], and Chengyu Song[†]

[†]UC Riverside [‡]Cornell University
[†]{jwang131,wsong008}@ucr.edu, {heng.csong}@cs.ucr.edu [‡]yd375@cornell.edu

Registered Report: DATAFLOW Towards a Data-Flow-Guided Fuzzer

Adrian Herrera
ANU & DST
adrian.herrera@anu.edu.au

Mathias Payer
EPFL
mathias.payer@epfl.ch

Anthony L. Hosking
ANU
anthony.hosking@anu.edu.au

Abstract—Recent advances in fuzz testing several forms of feedback mechanisms, fact that for a large range of programs a coverage alone is insufficient to reveal complex behaviors. We present a new representation looking for a match between the structure and adaptability to the testing. In particular, we believe that data flow (DDF) represents a good candidate for this information embedded by this data structure useful to find vulnerable constructs by means of graph overlap with the control flow of the fuzzer to trigger. Since some portions of the graph overlap with the control flow of the fuzzer to trigger, the additional information only “interesting” data-flow dependencies the fuzzer to visit the code in a distinct standard methodologies.

To test these observations, in this paper we present DDFuzz, a new approach that rewards the fuzzer with code coverage information, but also in the data dependency graph are highlighted. The adoption of data dependency in coverage-guided fuzzing is a promising solution to discover bugs that would otherwise remain standard coverage approaches. This is the first systematic study on these metrics in fuzzing. We discuss the concept of coverage metrics in fuzzing and compare different studies on these metrics. We present the LAVA-M dataset, and of 221 binaries. We have limited resources: metric has its unique of branches (this vulnerability slam coverage also explore combined cross-seeding, and the fuzzing based approach of binaries in the CGC that combines fuzzing time, our approach is

Coverage-guided fuzzing is a most common technique, which decides essential parameter of results. While there are many different coverage metrics, it is unclear whether it is superior to all the first systematic study on these metrics in fuzzing. We discuss the concept of coverage metrics in fuzzing and compare different studies on these metrics. We present the LAVA-M dataset, and of 221 binaries. We have limited resources: metric has its unique of branches (this vulnerability slam coverage also explore combined cross-seeding, and the fuzzing based approach of binaries in the CGC that combines fuzzing time, our approach is

Abstract—Coverage-guided greybox fuzzers rely on feedback derived from control-flow coverage to explore a target program and uncover bugs. This is despite control-flow feedback offering only a coarse-grained approximation of program behavior. Data flow intuitively more accurately characterizes program behavior. Despite this advantage, fuzzers driven by data-flow coverage have received comparatively little attention, appearing mainly when heavyweight program analyses (e.g., taint analysis, symbolic execution) are used. Unfortunately, these more accurate analyses incur a high run-time penalty, impeding fuzzer throughput. Lightweight data-flow alternatives to control-flow fuzzing remain unexplored.

We present DATAFLOW, a greybox fuzzer driven by lightweight data-flow profiling. Whereas control-flow edges represent the order of operations in a program, data-flow edges capture the dependencies between operations that produce data values and the operations that consume them. Indeed, there may be no control dependence between these operations. As such, data-flow coverage captures behaviors not visible as control flow and intuitively discovers more or different bugs. Moreover, we establish a framework for reasoning about data-flow coverage, allowing the computational cost of exploration to be balanced with precision.

We perform a preliminary evaluation of DATAFLOW, comparing fuzzers driven by control flow, taint analysis (both approximate and exact), and data flow. Our initial results suggest that, so far, pure coverage remains the best coverage metric for discovering bugs in our targets (72% of them). However, data-flow coverage does show promise in targets where control flow is decoupled from semantics (e.g., parsers). Further evaluation and analysis on a wider range of targets is required.

1 Introduction

Greybox fuzzing is a technique that has been widely used in security research. It involves running a target program with random inputs (e.g., Trail of Bits Challenge (CGC), gr) to be more effective in symbolic execution and

USENIX Association

1. INTRODUCTION

Fuzzers are an indispensable tool in the software-testing toolbox. The idea of fuzzing—to test a target program by subjecting it to a large number of randomly-generated inputs—can be traced back to an assignment in a graduate Advanced Operating Systems class [1]. These fuzzers were relatively primitive (compared to a modern fuzzer): they simply fed a randomly-generated input to the target, failing the test if the target crashed or hung. They did not model program or input structure, and could only observe the input/output behavior of the target. In contrast, modern fuzzers use sophisticated

program analyses to model program and input structure, and continuously gather dynamic information about the target.

Leveraging dynamic information drives fuzzer efficiency. For example, *coverage-guided greybox fuzzers*—perhaps the most widely-used class of fuzzer—track code paths executed by the target.¹ This allows the fuzzer to focus its mutations on inputs reaching new code. Intuitively, a fuzzer cannot find bugs in code never executed, so maximizing the amount of code executed should maximize the number of bugs found. Code coverage serves as an approximation of program behavior, and expanding code coverage implies exploring program behaviors.

Coverage-guided greybox fuzzers are now pervasive. Their success [2] can be attributed to one fuzzer in particular: American Fuzzy Lop (AFL) [3]. AFL is a greybox fuzzer that uses lightweight instrumentation to track edges covered in the target's control-flow graph (CFG). A large body of research has built on AFL [4–12]. While improvements have been made, most fuzzers still default to edge coverage as an approximation of program behavior. *Is this the best we can do?*

In some targets, control flow offers only a coarse-grained approximation of program behavior. This includes targets whose control structure is decoupled from its semantics (e.g., LR parsers generated by `yacc`) [13]. Such targets require data-flow coverage [13–17]. Whereas control flow focuses on the order of operations in a program (i.e., branch and loop structures), data flow instead focuses on how variables (i.e., data) are defined and used [14]: indeed, there may be no control dependence between variable definition and use sites (see §III for details).

In fuzzing, data flow typically takes the form of *dynamic taint analysis* (DTA). Here, the target's input data is tainted at its definition site and tracked as it is accessed and used at runtime. Unfortunately, accurate DTA is difficult to achieve and expensive to compute (e.g., prior work has found DTA is expensive [18, 19] and its accuracy highly variable across implementations [18, 20]). Moreover, several real-world programs fail to compile under DTA, increasing deployability concerns. Thus, most widely-deployed greybox fuzzers (e.g., AFL [3], libFuzzer [21], and honggfuzz [22]) eschew DTA in favor of higher fuzzing throughput.

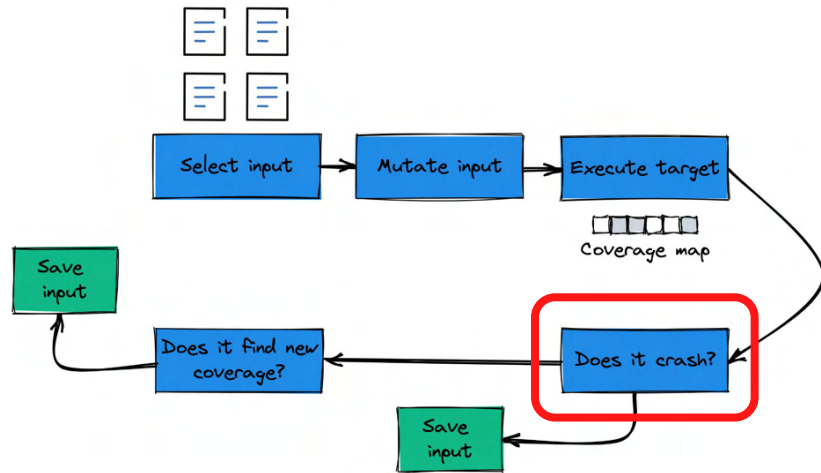
While lightweight alternatives to DTA exist (e.g., REQUICKER [23], GREYONE [19]), the full potential of control- vs. data-flow based fuzzer coverage metrics have not yet been thoroughly explored. To support this exploration, we

¹Miller et al.'s original fuzzer [1] is now known as a *blackbox fuzzer*, because it has no knowledge of the target's internals.

Greybox fuzzing

Does it crash?

- Classic memory-safety violation
 - SIGSEGV

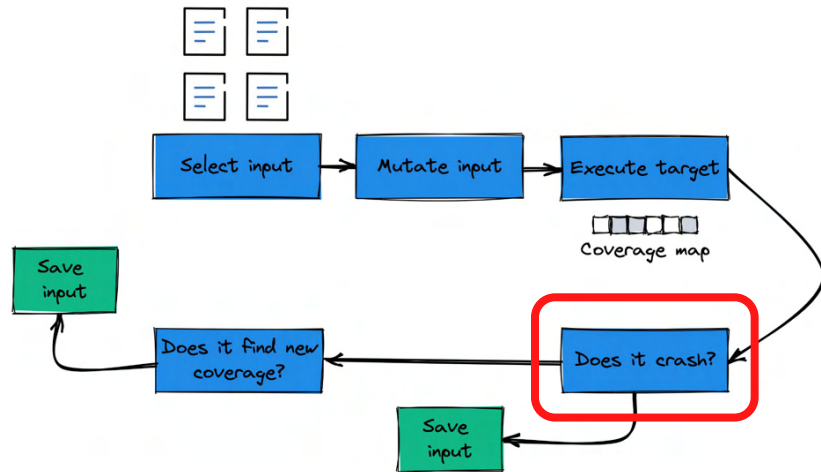


Greybox fuzzing

Does it crash?

- Classic memory-safety violation
 - SIGSEGV

What about other bug types?



Sanitization

- Allow for additional security policies to be defined and checked at runtime
- Typically compiler-based (e.g., LLVM)
 - But don't have to be

SoK: Sanitizing for Security

Dokyoung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

University of California, Irvine
{dokyungs, jlettner, rajasekp, yeoul, stijnv, perl, franz}@uci.edu

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Sushant Dinesh
Purdue University

Nathan Burrow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL

HexType: Efficient Detection of Type Confusion Errors for C++

Yuseok Jeon
Purdue University
jeon41@purdue.edu

Priyam Biswas
Purdue University
biswas12@purdue.edu

Scott Carr
Purdue University
carr27@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Abstract—The C and C++ remain insecure yet remain resort to a multi-pronged set of defenses. These include analysis, dynamic bug finders, and security issues. Specifically, the security vulnerabilities and compatibility properties

Abstract—Analyzing the security of a vast number of sanitizers with an overview of sanitizers with security issues. Specifically, the security vulnerabilities and compatibility properties

Abstract—Analyzing the security of a vast number of sanitizers with an overview of sanitizers with security issues. Specifically, the security vulnerabilities and compatibility properties

Abstract—Analyzing the security of a vast number of sanitizers with an overview of sanitizers with security issues. Specifically, the security vulnerabilities and compatibility properties

ABSTRACT
Type confusion, often combined with use-after-free, is the main attack vector to compromise modern C++ software like browsers or virtual machines.

Typescasting is a core principle that enables modularity in C++. For performance, most types are only checked statically, i.e., the check only tests if a cast is allowed for the given type hierarchy, ignoring the actual runtime type of the object. Using an object of an incompatible base type instead of a derived type results in type confusion. Attackers abuse such type confusion issues to attack popular software products including Adobe Flash, PHP, Google Chrome, or Firefox.

We propose to make all type checks explicit, replacing static checks with full runtime type checks. To minimize the performance impact of our mechanism HexType, we develop both low-overhead data structures and compiler optimizations. To maximize detection coverage, we handle specific object allocation patterns, e.g., placement new or reinterpret_cast which are not handled by other mechanisms.

Our prototype results show that, compared to prior work, HexType has at least 1.1 – 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2000 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++.

CCS CONCEPTS
Security and privacy → Systems security: Software and application security;

KEYWORDS
Type confusion, Bad casting, Type safety, Typescasting, Static_cast, Dynamic_cast, Reinterpret_cast

1 INTRODUCTION
C++ is well suited for large software projects as it combines high level modularity and abstraction with low level memory access and

performance. Common examples of C++ software include Google Chrome, MySQL, the Oracle Java Virtual Machine, and Firefox, all of which form the basis of daily computing uses for end-users.

The runtime performance efficiency and backwards compatibility to C come at the price of safety: enforcing memory and type safety is left to the programmer. This lack of safety leads to type confusion vulnerabilities that can be abused to attack programs allowing the attacker to gain full privileges of these programs. Type confusion vulnerabilities are a challenging mixture between lack of type and memory safety.

Generally, type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts.

For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [6, 14, 28, 38], and (ii) mechanisms that leverage disjoint metadata [15, 21]. First, solutions that rely on the existing object format have the advantage of avoiding expensive runtime object tracking to maintain disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their vtable pointer. As most software mixes both polymorphic and non-polymorphic objects, these solutions are limited in practice — either developers must manually blacklist unsupported classes or programs end up having unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

Sanitization

- Allow for additional security policies to be defined and checked at runtime
- Typically compiler-based (e.g., LLVM)
 - But don't have to be

What can we check for?

SoK: Sanitizing for Security

Dokyung Song, Julian Lettner, Prabhu Rajasekaran,
Yeoul Na, Stijn Volckaert, Per Larsen, Michael Franz

University of California, Irvine
{dokyungs, jlettner, rajasekp, yeoul, stjnov, perl, franz}@uci.edu

RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization

Sushant Dinesh
Purdue University

Nathan Burrow
Purdue University

Dongyan Xu
Purdue University

Mathias Payer
EPFL

HexType: Efficient Detection of Type Confusion Errors for C++

Yuseok Jeon
Purdue University
jeon41@purdue.edu

Priyam Biswas
Purdue University
biswas12@purdue.edu

Scott Carr
Purdue University
carr27@purdue.edu

Byoungyoung Lee
Purdue University
byoungyoung@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Abstract—The C and C++ remain insecure yet remain resort to a multi-pronged set of defenses. These include analysis, dynamic bug finders, and security issues. Specifically, the security vulnerabilities and compatibility properties

I. IN C and C++ remain the systems software such as libraries, and browsers. A and leave the program hardware. On the flip side, every memory access is undefined behavior, etc. In short of meeting these req make the code vulnerable. At the same time, more sophisticated [1]–[4] tions such as Address Spo and Data Execution Preven as Return-Oriented Progr such as function pointers control-flow of the progr Data-Oriented Programmi can be invoked on legal c program by corrupting on As a first line of defin analysis tools to identify is deployed in production program analysis, dynamic Static tools analyze the I results that are conservative of the code [5]–[9]. In o often called “sanitizers” and output a precise analy Sanitizers are now in a many vulnerability discove and critical role in finding not well-understood, whi

I. INTRODUCTION

Most software for commodity sy even developers for such systems source libraries. Even on Linux, vi as Skype, the Google Hangouts p closed source. Consequently, users the mercy of third-parties to det security issues. While mitigations s Stack Canaries [3], or CFI [4], [5] they cannot pinpoint the underlyi To discover memory errors d combine a feedback-guided fuzzer guide information about the executi such as AFL [6] leverage coverage tools such as Address Sanitizer (accesses for possible violations. TI as compiler-passes to instrument th resulting in low runtime overhead. I many software testing either: (i) res resulting in shallow coverage close (ii) rely on dynamic binary transl the binary at prohibitively high run for AFL fuzzing in QEMU mode use unsound static rewriting based

KEYWORDS

Type confusion, Bad casting, Type safety, Typecasting, Static_cast, Dynamic_cast, Reinterpret_cast

Abstract—Analyzing the security currently important for end-users, on third-party libraries. Such analys ability discovery techniques, most n enabled. The current state of the i sanitization to binaries is dynamic b prohibitive performance overhead. TI binary rewriting, cannot fully reco and hence has difficulty modifying bl for fuzzing or to add security check The ideal solution for binary securi rewriter that can intelligently add c as if it were inserted at compile t requires an analysis to statically dia and scalars, a problem known to be case. We show that recovering this practice for the most common cla de-ke, position independent code. I we develop RetroWrite, a binar to support American Fuzzy Lop (A (ASoL), and show that it can ad vance while retaining precision. Bin guided fuzzing using RetroWrite i to compiler-instrumented binaries i QEMU-based instrumentation by bugs. Our implementation of binary- later than Valgrind's memcheck, the memory checker, and detects 80% n

ABSTRACT

Type confusion, often combined with use-after-free, is the main attack vector to compromise modern C++ software like browsers or virtual machines. Typecasting is a core principle that enables modularity in C++. For performance, most types are only checked statically, i.e., the check only tests if a cast is allowed for the given type hierarchy, ignoring the actual runtime type of the object. Using an object of an incompatible base type instead of a derived type results in type confusion. Attackers abuse such type confusion issues to attack popular software products including Adobe Flash, PHP, Google Chrome, or Firefox. We propose to make all type checks explicit, replacing static checks with full runtime type checks. To minimize the performance impact of our mechanism HexType, we develop both low-overhead data structures and compiler optimizations. To maximize detection coverage, we handle specific object allocation patterns, e.g., placement new or reinterpret_cast which are not handled by other mechanisms. Our prototype results show that, compared to prior work, HexType has at least 1.1 – 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2006 benchmarks with overhead, we show a 33.4 times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++.

CCS CONCEPTS

Security and privacy → Systems security: Software and application security;

KEYWORDS

Type confusion, Bad casting, Type safety, Typecasting, Static_cast, Dynamic_cast, Reinterpret_cast

1 INTRODUCTION

C++ is well suited for large software projects as it combines high level modularity and abstraction with low level memory access and 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 from the publisher. For all other uses, contact the publisher. Copyright 2017, Oct 30–Nov 3, 2017, Dallas, TX, USA. © 2017 Copyright held by the owner(s). Publication rights licensed to ACM. ISBN 978-1-4503-4517-1/17/10... \$15.00 DOI: <http://dx.doi.org/10.1145/3333998.3334062>

performance. Common examples of C++ software include Google Chrome, MySQL, the Oracle Java Virtual Machine, and Firefox, all of which form the basis of daily computing uses for end-users. The runtime performance efficiency and backwards compatibility to C come at the price of safety: enforcing memory and type safety is left to the programmer. This lack of safety leads to type confusion vulnerabilities that can be abused to attack programs allowing the attacker to gain full privileges of these programs. Type confusion vulnerabilities are a challenging mixture between lack of type and memory safety.

Generally, type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts. For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [6, 14, 28, 38], and (ii) mechanisms that leverage disjoint metadata [15, 21]. First, solutions that rely on the existing object format have the advantage of avoiding expensive runtime object tracking to maintain disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their vtable pointer. As most software mixes both polymorphic and non-polymorphic objects, these solutions are limited in practice – either developers must manually blacklist unsupported classes or programs end up having unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not

Sanitization

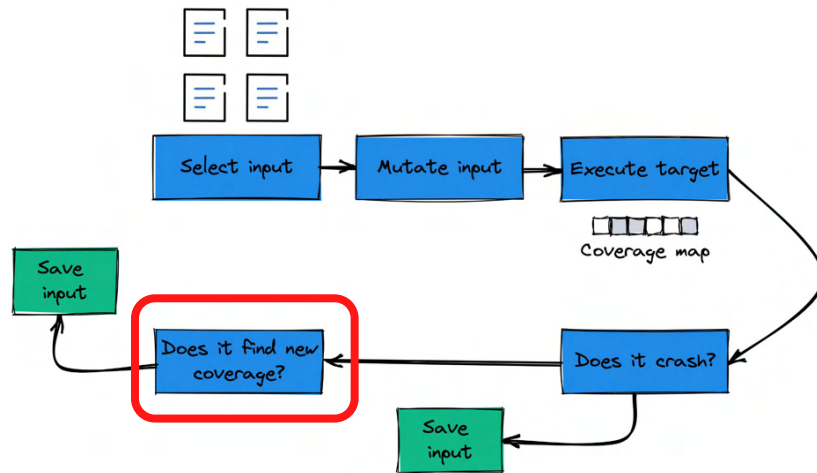
Anything we can encode as an **invariant**

- Address Sanitizer (ASan)
- Undefined behavior Sanitizer (UBSan)
- Memory Sanitizer (MSan)
- LeakSanitizer (LSan)
- ThreadSanitizer (TSan)

Greybox fuzzing

Does it find new coverage?

- Save input
- Return to start



What about...

- Non-file, non-*nix fuzzing
 - E.g., network services, OS kernel, IoT, ...
- Writing a harness
 - The fuzzer has to start somewhere
- Overcoming “roadblocks”
 - E.g., complex conditionals

What about...

- Non-file, non-*nix fuzzing
 - E.g., network services, OS kernel, IoT, ...
- Overcoming “roadblocks”
 - E.g., complex conditionals

*nix file fuzzing

- Primary focus of academic research
- Assumes an “obvious” entry point
 - AFL-style fuzzing: `main + fread`
 - libFuzzer: dedicated `LLVMFuzzerTestOneInput`
- Commonly assumes source code

*nix file fuzzing

- Primary focus of academic research
- Assumes an “obvious” entry point
 - AFL-style fuzzing: `main + fread`
 - libFuzzer: dedicated `LLVMFuzzerTestOneInput`
- Commonly assumes source code

What is the entry point for a network service / OS kernel / IoT device? 🤔

Network apps

Challenges

- State
- Setup/teardown connection cost
- What is “coverage”?

Solutions

- Snapshots
 - No need to start from scratch each time
- Annotate/infer states

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3}, Ali Dav
¹ Beijing K
Ins

³ School of Cyber
[zhengyaowen,zhuangsho

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerabil Fuzzing is an effective software tes nerability discovery. In this work, we first high-throughput greybox fuzzer AFL addresses two fundamental pr First, it addresses compatibility issues POSIX-compatible firmware that can emulator. Second, it addresses the f caused by system-mode emulation called augmented process emulation mode emulation and user-mode em augmented process emulation provid system-mode emulation and high th emulation. Our evaluation results sho fully functional and capable of findi ties in IoT programs; (2) the through average 8.2 times higher than system (fuzzing); and (3) FIRM-AFL is able ti ties much faster than system-mode e and is able to find 0-day vulnerabilities

1 Introduction

The security impact of IoT devices or By 2020, the number of connected IoT number of people [110]. This creates i surface leaving almost everything at the hackers leverage the lack of sec create large botnets (e.g., Mirai, VPN malware attacks exploit the vulnera to penetrate into the IoT devices. As defenders to discover vulnerabilities then before attackers.

¹This work is done while visiting Univer
²Corresponding author

USENIX Association

Abstract

OS fuzzers primarily tween the OS kernel a rity vulnerabilities. TI lutionary OS fuzzers diversity of their seed generating good seed as the behavior of ea the OS kernel state c system calls. Therefor often rely on hand-c sequences of system i process. Unfortunately the diversity of the se fore limits the effecti

In this paper, we d ecy for distilling see traces of real-world p dependencies across i ages light-weight sta dependencies across i We designed and extension to Syskall fuzzer for the Linux taining 2.8 million r real-world programs, over 14,000 calls to the code coverage. Usin sequences, MoonShi achieved code covera average. MoonShine in the Linux kernel th

1 Introduction

Security vulnerability after-free inside open ularly dangerous a completely compromise a popular technique fixing such critical s fuzzers focus primari cause as it is one of the OS kernel and is

CCS Concepts • rity • Software i cation and valid

Keywords: Testin

ACM Reference Fi
Sergej Schumilo, C basi, and Thorsten Incremental Snapsh pair Systems (Earl New York, NY, USA,



This work is licensed under

EuroSys '22, April 3–6,
© 2022 Copyright held
ACM ISBN 978-1-60959
https://doi.org/10.1146

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz³

¹Ruhr-Universität Bochum, ²Vrije Universiteit Amsterdam
³CISPA Helmholtz Center for Information Security

SnapFuzz: High-Throughput Fuzzing of Network Applications

Anastasios Andronidis
Imperial College London
London, United Kingdom
a.andronidis@imperial.ac.uk

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts.

In this paper, we present SnapFuzz, a novel fuzzing framework for network applications. SnapFuzz offers a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using SnapFuzz, we fuzzed five popular networking applications: LightFTP, TinyDTLS, Dnsmaas, LIVE555 and Dcmppp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, SnapFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; • Security and privacy → Systems security.

KEYWORDS

Fuzzing, network protocol implementations, stateful applications

ACM Reference Format

Anastasios Andronidis and Cristian Cadar: 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance,

Google has discovered over 25,000 bugs in their products and over 22,000 bugs in open-source code using greybox fuzzing [118].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, network protocol implementations, is difficult to fuzz. There are two main difficulties: the fact that in-depth testing of such applications needs to be aware of the network protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

There are two main approaches for testing such software in a meaningful way. One approach, adopted by Google's OSS-Fuzz, is to write unit-level test drivers that interact with the software via its API [11]. While such an approach can be effective, it requires significant manual effort, and does not perform system-level testing where an actual server instance interacts with actual clients.

A second approach, used by AFLNet [36], performs system-level testing by starting actual server and client processes, and generating random message exchanges between them which nevertheless follow the underlying network protocol. Furthermore, it does so without needing a specification of the protocol, but rather by using a corpus of real message exchanges between server and clients. AFLNet's approach has significant advantages, requiring less manual effort and performing end-to-end testing at the protocol level.

While AFLNet makes important advances in terms of fuzzing network protocols, it has two main limitations. First, it requires users to add or configure various time delays in order to make sure the protocol is followed, and to write clean-up scripts to reset the state across fuzzing iterations. Second, it has poor fuzzing performance, caused by asynchronous network communication, various time delays, and expensive file system operations, among others.

SnapFuzz addresses both of these challenges through a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the forkserver placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

2 FROM AFL TO AFLNET TO SNAPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing on their internal architecture and performance implications, and then contribute an overview of SnapFuzz's architecture and main contributions.

OS kernel

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

- kCOV + kASan
- Hypervisor + PMU
- Seeds = syscall traces

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3}, Ali Dav
¹ Beijing K
Ins

³ School of Cyber
[zhengyaowen,zhuahong]

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerabil Fuzzing is an effective software tes nerability discovery. In this work, we first high-throughput greybox fuzzer AFL addresses two fundamental pr First, it addresses compatibility issues POSIX-compatible firmware that can emulator. Second, it addresses the f caused by system-mode emulation called augmented process emulation mode emulation and user-mode em augmented process emulation provid system-mode emulation and high th emulation. Our evaluation results sho fully functional and capable of findi ties in IoT programs; (2) the through average 8.2 times higher than system fuzzing; and (3) FIRM-AFL is able ti ties much faster than system-mode e and is able to find 0-day vulnerabilities

1 Introduction

The security impact of IoT devices or By 2020, the number of connected IoT number of people [10]. This creates i surface leaving almost everything o the hackers leverage the lack of se create large botnets (e.g., Mirai, VPS malware attacks exploit the vulnera to penetrate into the IoT devices. As defenders to discover vulnerabilities then before attackers.

¹This work was done while visiting Univer
²Corresponding author

USENIX Association

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily tween the OS kernel a rity vulnerabilities. TI lutionary OS fuzzers diversity of their seed generating good seed as the behavior of ea the OS kernel state c system calls. Therefor often rely on hand-co sequences of system i process. Unfortunately the diversity of the se fore limits the effecti In this paper, we d egy for distilling see traces of real-world p dependencies across i ages light-weight sta dependencies across i

1 Introduction

We designed and extension to Syskall fuzzer for the Linux taining 2.8 million r real-world programs, over 14,000 calls w code coverage. Usi sequences, MoonShi achieved code covera average. MoonShine in the Linux kernel th

1 Introduction

Security vulnerability after-free inside oper ticularly dangerous a completely compromise a popular technique fixing such critical s fuzzers focus primari face as it is one of the OS kernel and us

Keywords: Testin
ACM Reference Fi
Sergej Schumilo, Co
basi, and Thorsten
Incremental Snapsh
paler Systems (Earl
New York, NY, USA,

CC BY
This work is licensed under

EuroSys '22, April 3–6,
© 2022 Copyright held
ACM ISBN 978-1-60959
https://doi.org/10.1145

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz³
¹Ruhr-Universität Bochum, ²Vrije Universiteit Amsterdam
³CISPA Helmholtz Center for Information Security

SnappFuzz: High-Throughput Fuzzing of Network Applications

Anastasios Andronidis
Imperial College London
London, United Kingdom
a.andronidis@imperial.ac.uk

Cristian Cadar
Imperial College London
London, United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts. In this paper, we present SnappFuzz, a novel fuzzing framework for network applications. SnappFuzz offers a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts. Using SnappFuzz, we fuzzed five popular networking applications: LightFTP, TinyDTLS, Dnsmsg, LIVEDNS and Dnspsp. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, SnappFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; • Security and privacy → Systems security.

KEYWORDS

Fuzzing, network protocol implementations, stateful applications

ACM Reference Format:

Anastasios Andronidis and Cristian Cadar. 2022. SnappFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 19–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance,

Google has discovered over 25,000 bugs in their products and over 22,000 bugs in open-source code using greybox fuzzing [18].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, network protocol implementations, is difficult to fuzz. There are two main difficulties: the fact that in-depth testing of such applications needs to be aware of the network protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

There are two main approaches for testing such software in a meaningful way. One approach, adopted by Google's OSS-Fuzz, is to write unit-level test drivers that interact with the software via its API [11]. While such an approach can be effective, it requires significant manual effort, and does not perform system-level testing where an actual server instance interacts with actual clients.

A second approach, used by AFLNet [36], performs system-level testing by starting actual server and client processes, and generating random message exchanges between them which nevertheless follow the underlying network protocol. Furthermore, it does so without needing a specification of the protocol, but rather by using a corpus of real message exchanges between server and clients. AFLNet's approach has significant advantages, requiring less manual effort and performing end-to-end testing at the protocol level.

While AFLNet makes important advances in terms of fuzzing network protocols, it has two main limitations. First, it requires users to add or configure various time delays in order to make sure the protocol is followed, and to write clean-up scripts to reset the state across fuzzing iterations. Second, it has poor fuzzing performance, caused by asynchronous network communication, various time delays, and expensive system operations, among others.

SnappFuzz addresses both of these challenges through a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the fuzzer's placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

2 FROM AFL TO AFLNET TO SNAPPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing on their internal architecture and performance implications, and then contribute an overview of SnappFuzz's architecture and main contributions.

IoT

Challenges

- Measuring coverage
- Performance
- Seeds?

Solutions

- QEMU (slow / incomplete)
- Avatar² orchestration

FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng^{1,2,3}, Ali Dav
¹ Beijing K
Ins

³ School of Cyber
[zhengyaowen,zhuahong]

Abstract

Cyber attacks against IoT devices an attacks exploit software vulnerabil Fuzzing is an effective software tes nerability discovery. In this work, we first high-throughput greybox fuzzer AFL addresses two fundamental pr First, it addresses compatibility issues POSIX-compatible firmware that can emulator. Second, it addresses the f caused by system-mode emulation called augmented process emulation mode emulation and user-mode em augmented process emulation provid system-mode emulation and high th emulation. Our evaluation results sho fully functional and capable of findi ties much faster than system-mode n and is able to find 0-day vulnerabili

1 Introduction

The security impact of IoT devices or By 2020, the number of connected IoT number of people [10]. This creates i surface leaving almost everybody at the hackers leverage the lack of sec create large botnets (e.g., Mirai, VPN malware attacks exploit the vulnera to penetrate into the IoT devices. As defenders to discover vulnerabilities then before attackers.

¹This work is done while visiting Univer
²Corresponding author

USENIX Association

MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily tween the OS kernel a rity vulnerabilities. TI lutionary OS fuzzers diversity of their seed generating good seed as the behavior of ea the OS kernel state c system calls. Therefor often rely on hand-co sequences of system i process. Unfortunately the diversity of the se fore limits the effecti

In this paper, we d ecy for distilling see traces of real-world p dependencies across i ages light-weight sta dependencies across i We designed and extension to Syskall fuzzer for the Linux taining 2.8 million r real-world programs, over 14,000 calls w code coverage. Use sequences, MoonShi achieved code covera average. MoonShine in the Linux kernel th

1 Introduction

Security vulnerability after-free inside open icularly dangerous a completely comprom a popular technique i fixing such critical s fuzzers focus primar because it is one of the OS kernel and is

CCS Concepts • rity • Software i cation and valid

Keywords: Testin

ACM Reference Fi
Sergej Schumilo, Co basi, and Thorsten Incremental Snapsh pair Systems (Eand New York, NY, USA,



EuroSys '22, April 3–6,
© 2022 Copyright held
ACM ISBN 978-1-60959
https://doi.org/10.1145

Nyx-Net: Network Fuzzing with Incremental Snapshots

Sergej Schumilo¹, Cornelius Aschermann¹, Andrea Jemmett², Ali Abbasi¹, and Thorsten Holz³
¹Ruhr-Universität Bochum, ²Vrije Universiteit Amsterdam
³CISPA Helmholtz Center for Information Security

SnappFuzz: High-Throughput Fuzzing of Network Applications

Anastasios Andronidis
Imperial College London,
London, United Kingdom
a.andronidis@imperial.ac.uk

Cristian Cadar
Imperial College London,
London, United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

In recent years, fuzz testing has benefited from increased computational power and important algorithmic advances, leading to systems that have discovered many critical bugs and vulnerabilities in production software. Despite these successes, not all applications can be fuzzed efficiently. In particular, stateful applications such as network protocol implementations are constrained by a low fuzzing throughput and the need to develop complex fuzzing harnesses that involve custom time delays and clean-up scripts. In this paper, we present SnappFuzz, a novel fuzzing framework for network applications. SnappFuzz offers a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, snapshots the target at the latest point at which it is safe to do so, speeds up file operations by redirecting them to a custom in-memory filesystem, and removes the need for many fragile modifications, such as configuring time delays or writing clean-up scripts.

Using SnappFuzz, we fuzzed five popular networking applications: LightFTP, TinyDTLS, Dnsmsg, LIVEDNS and Dnsmpg. We report impressive performance speedups of 62.8 x, 41.2 x, 30.6 x, 24.6 x, and 8.4 x, respectively, with significantly simpler fuzzing harnesses in all cases. Due to its advantages, SnappFuzz has also found 12 extra crashes compared to AFLNet in these applications.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; • Security and privacy → Systems security.

KEYWORDS

Fuzzing, network protocol implementations, stateful applications

ACM Reference Format:

Anastasios Andronidis and Cristian Cadar: 2022. SnappFuzz: High-Throughput Fuzzing of Network Applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea. ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3533767.3534376

1 INTRODUCTION

Fuzzing is an effective technique for testing software systems, with popular fuzzers such as AFL and LibFuzzer having found thousands of bugs in both open-source and commercial software. For instance,

Google has discovered over 25,000 bugs in their products and over 22,000 bugs in open-source code using greybox fuzzing [18].

Unfortunately, not all software can benefit from such fuzzing campaigns. One important class of software, network protocol implementations, is difficult to fuzz. There are two main difficulties: the fact that in-depth testing of such applications needs to be aware of the network protocol they implement (e.g., FTP, DICOM, SIP), and the fact that they have side effects, such as writing data to the file system or exchanging messages over the network.

There are two main approaches for testing such software in a meaningful way. One approach, adopted by Google's OSS-Fuzz, is to write unit-level test drivers that interact with the software via its API [11]. While such an approach can be effective, it requires significant manual effort, and does not perform system-level testing where an actual server instance interacts with actual clients.

A second approach, used by AFLNet [30], performs system-level testing by starting actual server and client processes, and generating random message exchanges between them which nevertheless follow the underlying network protocol. Furthermore, it does so without needing a specification of the protocol, but rather by using a corpus of real message exchanges between server and clients. AFLNet's approach has significant advantages, requiring less manual effort and performing end-to-end testing at the protocol level.

While AFLNet makes important advances in terms of fuzzing network protocols, it has two main limitations. First, it requires users to add or configure various time delays in order to make sure the protocol is followed, and to write clean-up scripts to reset the state across fuzzing iterations. Second, it has poor fuzzing performance, caused by asynchronous network communication, various time delays, and expensive file system operations among others.

SnappFuzz addresses both of these challenges through a robust architecture that transforms slow asynchronous network communication into fast synchronous communication, speeds up file operations and removes the need for clean-up scripts via an in-memory filesystem, and improves other aspects such as delaying and automating the fuzzer's placement, correctly handling signal propagation and eliminating developer-added delays.

These improvements significantly simplify the construction of fuzzing harnesses for network applications and dramatically improve fuzzing throughput in the range of 8.4 x to 62.8 x (mean 30.6 x) for a set of five popular server benchmarks.

2 FROM AFL TO AFLNET TO SNAPPFUZZ

In this section, we first discuss how AFL and AFLNet work, focusing on their internal architecture and performance implications, and then contribute an overview of SnappFuzz's architecture and main contributions.

Fuzzer harnessing

Relates to previous research

Can we automatically synthesize a harness?

- Or at least help us develop one...

WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning

Jinho Jung, Stephen Tong, Ho
Georgia Institute of Te

Abstract—Fuzzing is an emerging technique to automatically validate programs and uncover bugs. It has been used to test many programs and has found thousands of vulnerabilities. However, existing fuzzing efforts are mainly around Unix-like systems, as Windows imposes unique challenges for fuzzing: a closed-source ecosystem, the heavy use of interfaces and the lack of fast process cloning machine

In this paper, we propose two solutions to address these challenges. Windows fuzzing faces. Our system, WINNIE, tries to synthesize a harness for the application, a simple that directly invokes target functions, based on sample. It then tests the harness, instead of the original application, using an efficient implementation of fork on Windows. Using these techniques, WINNIE can bypass irrelevant code to test logic deep within the application. We used WINNIE to fuzz 59 closed-source Windows binaries, and it successfully found 22 vulnerabilities for all of them. In our evaluation, we support 2.2x more programs than existing fuzzers could, and identified 3.9x more program states and 26.6x faster execution. In total, WINNIE found 61 out of 32 Windows binaries.

1. INTRODUCTION

Fuzzing is an emerging software-testing technique that automatically validating program functionalities and security vulnerabilities [12]. It randomly mutates inputs to generate a large corpus and feeds each in program. It monitors the execution for abnormal behavior such as crashing, hanging, or failing security checks [5]. Fuzzing efforts have found thousands of vulnerabilities in source projects [12, 28, 52, 62]. There are continue to make fuzzing faster [4, 9, 53] and smarter [60, 6]

However, existing fuzzing techniques are mainly Unix-like OSes, and few of them work as well on Windows. Unfortunately, Windows applications suffer from bugs. Recent report shows that in the past 70% of all security vulnerabilities on Windows systems memory safety issues [43]. In fact, due to the distinct Windows operating system, its applications remain lucrative targets for malicious attackers [10, 17, 1]. Fuzzing popular fuzzing techniques to the Windows platform investigate common applications and state-of-the-art and identify three challenges of fuzzing applications: Windows: a predominance of graphical applications source ecosystem (e.g., third-party or legacy library lack of fast cloning machinery like fork on Unix-li

Network and Distributed Systems Security (NDSS) Symposium
21-23 February 2021, Virtual
ISBN 1-891562-66-5
<https://doi.org/10.14722/ndss.2021.24334>
<https://ndss.symposium.org>

JMPscore: Introspection for Binary-Only Fuzzing

Dominik Maier, Lukas Seidel

FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou
Google Inc.

Daniel Austin
Google Inc.

Vishwath Mohan
Google Inc.

Mathias Payer
EPFL

Abstract

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library's interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

1 Introduction

Modern software distributions like Debian, Ubuntu, and the Android Open Source Project (AOSP) are large and complex ecosystems with many different software components. Debian consists of a base system with hundreds of libraries, system services and their configurations, and a customized Linux kernel. Similarly, AOSP consists of the ART virtual machine, Google's support libraries, and several hundred third party components including open source libraries and vendor specific code. While Google has been increasing efforts

to fuzz test this code, e.g., OSS-Fuzz [35, 36], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [20, 27]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [44] or any of the more recent advances over AFL such as AFLfast [6], AFLGo [5], collAFL [19], Driller [37], VUzzer [31], T-Fuzz [28], QSYM [42], or Angora [8] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [24]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to *fuzzing libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These implicit dependencies between library calls are often mentioned in documentation but are generally not formally specified. Calling random exported functions with random arguments is unlikely to result in an efficient fuzzing campaign. For example, *libmpeg2* requires an allocated context that contains the current encoder/decoder configuration and buffer information. This context is passed to each subsequent library function. Random fuzzing input is unlikely to create this context and correctly pass it to later functions. Quite the contrary, it will generate a large number of false positive crashes when library dependencies are not enforced, e.g., the configuration function may set the length of the allocated decode buffer in the

Workshop on Binary Analysis
21 February 2021, Virtual
ISBN 1-891562-66-5
<https://doi.org/10.14722/binar.2021.24334>
<https://ndss.symposium.org>

Overcoming “roadblocks”

Program constraints that are hard to meet

Solutions

- Writebox fuzzing
- Concolic execution
- Rewrite the target 🤔

Driller: Augmenting Fuzzing Through Selective Symbolic Execution

Nick Stephens,
Jacopo Corbucci

{stephe

Abstract—Memory corruption vuln-
erabilities in software, which attack
unauthorized access to confidential
data, are a major security concern. They
are often difficult to exploit, and their
number is increasing rapidly, resulting in a
greater need for automated tools to
find them. In this paper, we present
Driller, a new fuzzer that combines
symbolic execution with fuzzing to
discover vulnerabilities in software.

We present Driller, a hybrid fuzzer
which leverages fuzzing and symbolic
execution to discover vulnerabilities in
software. Driller is designed to be used
as a drop-in replacement for existing
fuzzers. It performs symbolic execution
on the code paths that the fuzzer
explores, allowing it to discover
vulnerabilities that would otherwise
be missed. Driller has been evaluated
on a set of benchmarks, and it has
discovered several new vulnerabilities.

Despite efforts to increase the
against security flaws, vulnerability
commonplace. In fact, in recent
years, security vulnerabilities have
increased significantly. Furthermore,
despite the introduction of new
techniques for identifying and
mitigating security flaws, they continue
to account for a large portion of the
total number of security incidents in
the last year [14].

Permission to freely reproduce all or part
of this paper for noncommercial
purposes is granted provided that copies
bear this notice and the full citation
to the source, including the page
number, is included. Reproduction for
commercial purposes is strictly
prohibited without the prior written
consent of the Internet Society, the
first-named author, or the author's
employer if the paper was prepared
within the scope of employment.

Abstract

A major impediment to practical sym-
bolic execution is the cost of running
symbolic execution on the code paths
that the fuzzer explores. In this paper,
we present Driller, a hybrid fuzzer
which leverages fuzzing and symbolic
execution to discover vulnerabilities in
software. Driller is designed to be
used as a drop-in replacement for
existing fuzzers. It performs symbolic
execution on the code paths that the
fuzzer explores, allowing it to discover
vulnerabilities that would otherwise
be missed. Driller has been evaluated
on a set of benchmarks, and it has
discovered several new vulnerabilities.

1 Introduction

Symbolic execution was conceived
as a technique for finding bugs in
software [22]. While it has been
used for many years, it has only
recently become a practical technique
for finding bugs in software. This is
due to the development of new
techniques for performing symbolic
execution, such as the use of
heuristics to guide the search for
bugs [1, 2].

Despite the increase in popularity
of symbolic execution, it remains a
challenge for researchers to develop
tools that can find bugs in software
efficiently. In this paper, we present
Driller, a new fuzzer that combines
symbolic execution with fuzzing to
discover vulnerabilities in software.

Symbolic execution with SYMCC: Don't interpret, compile!

Sebastian Poelau

Aurélien Francillon

T-Fuzz: fuzzing by program transformation

Hui Peng
Purdue University
peng124@purdue.edu

Yan Shoshitaishvili
Arizona State University
yan@asu.edu

Mathias Payer
Purdue University
mathias.payer@nrcwvill.edu



Abstract—Fuzzing is a simple yet effective approach to discover
software bugs utilizing randomly generated inputs. However, it
is limited by coverage and cannot find bugs hidden in deep
execution paths of the program because the randomly generated
inputs fail complex sanity checks, e.g., checks on magic values,
checksums, or hashes.

To improve coverage, existing approaches rely on imprecise
heuristics or complex input mutation techniques (e.g., symbolic
execution or taint analysis) to bypass sanity checks. Our novel
method tackles coverage from a different angle: by removing
sanity checks in the target program. T-Fuzz leverages a coverage
guided fuzzer to generate inputs. Whenever the fuzzer can no
longer trigger new code paths, a light-weight, dynamic
tracing based technique detects the input checks that the fuzzer-
generated inputs fail. These checks are then removed from the
target program. Fuzzing then continues on the transformed
program, allowing the code protected by the removed checks
to be triggered and potential bugs discovered.

Fuzzing transformed programs to find bugs poses two chal-
lenges: (1) removal of checks leads to over-approximation and
false positives, and (2) even for true bugs, the crashing input on
the transformed program may not trigger the bug in the original
program. As an auxiliary post-processing step, T-Fuzz leverages
a symbolic execution-based approach to filter out false positives
and reproduce true bugs in the original program.

By transforming the program as well as mutating the input, T-
Fuzz covers more code and finds more true bugs than any existing
technique. We have evaluated T-Fuzz on the DARPA Cyber Grand
Challenge dataset, LAVA-M dataset and 4 real-world programs
(log4j, tiff, libtiff, magick and pdfcrowd). For the CGC dataset, T-Fuzz finds bugs in 166 binaries, Driller in
121, and AFL in 105. In addition, found 3 new bugs in previously-
fuzzed programs and libraries.

1. INTRODUCTION

Fuzzing is an automated software testing technique that
discovers bugs by providing randomly-generated inputs to the
program. It has been proven to be simple, yet effective [1], [2].
With the reduction of computational costs, fuzzing has become
increasingly useful for both hackers and software vendors, who
use it to discover new bugs/vulnerabilities in software. As such,

fuzzing has become a standard in software development to
improve reliability and security [3], [4].

Fuzzers can be roughly divided into two categories based on
how inputs are produced: generational fuzzers and mutational
fuzzers. Generational fuzzers, such as PROTON [5], SPIKE [6],
and PEACH [7], construct inputs according to some provided
format specification. By contrast, mutational fuzzers, including
AFL [8], honggfuzz [9], and zcrash [10], create inputs by ran-
domly mutating analyst-provided or randomly-generated seeds.
Generational fuzzing requires an input format specification,
which imposes significant manual effort to create (especially
when attempting to fuzz software on a large scale) or may
be infeasible if the format is not available. Thus, most recent
work in the field of fuzzing, including this paper, focuses on
mutational fuzzing.

Fuzzing is a dynamic technique. To find bugs, it must trigger
the code that contains these bugs. Unfortunately, mutational
fuzzing is limited by its coverage. Regardless of the muta-
tion strategy, whether it be a purely randomized mutation or
coverage-guided mutation, it is highly unlikely for the fuzzer
to generate inputs that can bypass complex sanity checks in
the target program. This is because, due to their simplicity,
mutational fuzzers are ignorant of the actual input format
expected by the program. This inherent limitation prevents
mutational fuzzers from triggering code paths protected by
sanity checks and finding “deep” bugs hidden in such code.

Fuzzers have adopted a number of approaches to better
mutate input to satisfy complex checks in a program. AFL [8],
for example, considered the state-of-art mutational fuzzer, uses coverage
to guide its mutation algorithm, with great success in real
programs [11]. To help bypass the sanity checks on magic
values in the input files, AFL uses coverage feedback to heuristi-
cally infer the values and positions of the magic values in the
input. Several recent approaches [12], [13], [14], [15] leverage
symbolic analysis or taint analysis to improve coverage by
generating inputs to bypass the sanity checks in the target
program. However, limitations persist — as we discuss in
our evaluation, state-of-the-art techniques such as AFL and
Driller find vulnerabilities in less than half of the programs
in a popular vulnerability analysis benchmarking dataset (the
challenge programs from the DARPA Cyber Grand Challenge).

Recent research into fuzzing techniques focuses on finding
new ways to generate and evaluate inputs. However, there
is no need to limit mutation to program inputs alone. In
fact, the program itself can be mutated to assist bug finding
in the fuzzing process. Following this intuition, we propose

Permission to freely reproduce all or part of this paper for noncommercial
purposes is granted provided that copies bear this notice and the full citation
to the source, including the page number, is included. Reproduction for
commercial purposes is strictly prohibited without the prior written
consent of the Internet Society, the first-named author, or the author's
employer if the paper was prepared within the scope of employment.

Whitebox fuzzing

Symbolic execution

- Translate expressions into **symbolic formulae**
- Program paths accumulate formulae into **constraints**
- Constraints are solved (via a **SAT / SMT solver**)

Challenges

- Expensive / slow
- Modeling “external environment”

Concolic fuzzing

Concolic = **con**crete + sym**bol**ic

- Symbolic values augmented with concrete values
- Can always fall back to concrete values

Solutions

- Angora: Treat solver as optimization problem
- SymCC: Compiles concolic executor into the binary
- JIGSAW: JIT compile constraints 🤖

What about...

- Directed fuzzers?
- Determining when we've "fuzzed enough"?
- Benchmarking fuzzers?

What about...

- Directed fuzzers?
- Determining when
- Benchmarking fuz

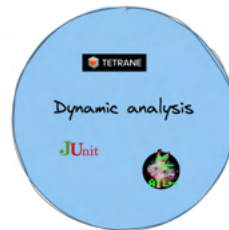


Conclusions

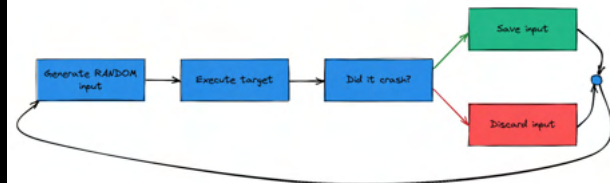
- Fuzzing research has progressed in leaps and bounds
 - No longer just “file-based + *nix-based”
- Still many open problems
- Balance between **performance** and **accuracy**



What is fuzzing?



Our first fuzzer

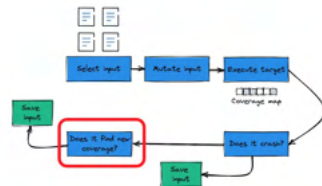


This is a classic **generational blackbox** fuzzer

Greybox fuzzing

Does it find new coverage?

- Save input
- Return to start



Grammar-based fuzzing

NAUTILUS: Fishing for Deep Bugs with Grammars



- Many targets (e.g., JavaScript interpreter) accept input described by a **context-free grammar (CFG)**
 - Highly structured
 - Blind mutation will destroy structure
- Leverage CFG in mutation
 - "Lift" inputs to parse tree
 - Mutate parse tree(s)
 - Lower parse tree back to file

Sanitization

- Allow for additional security policies to be defined and checked at runtime
- Typically compiler-based (e.g., LLVM), but don't have to be

What can we check for?



Conclusions

- Fuzzing research has progressed in leaps and bounds
 - No longer just "file-based + *nix-based"
- Still many open questions
- Balance between **performance** and **accuracy**