MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

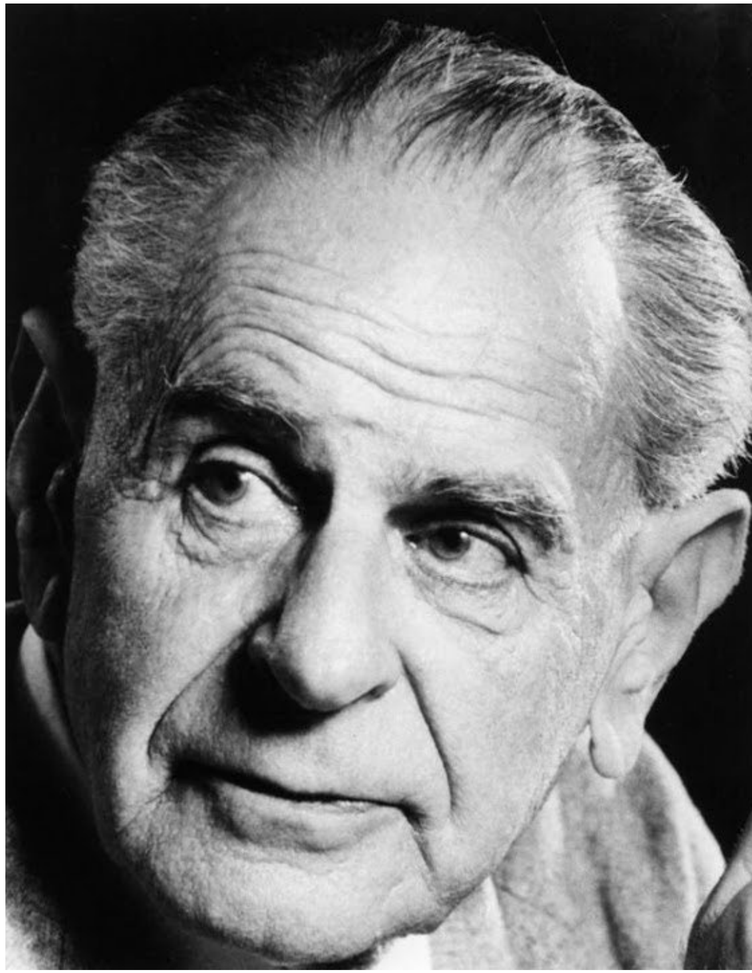
Benchmarks are our measure of progress. Or are they?

SCAN ME

Marcel Böhme
Max Planck Institute
for Security & Privacy

Scientific Enquiry as a Testing Problem

“I propose to replace [..] the question of the sources of our knowledge [e.g., how to identify the “best” scientific theory] by the entirely different question:
'How can we hope to detect and eliminate error?”'

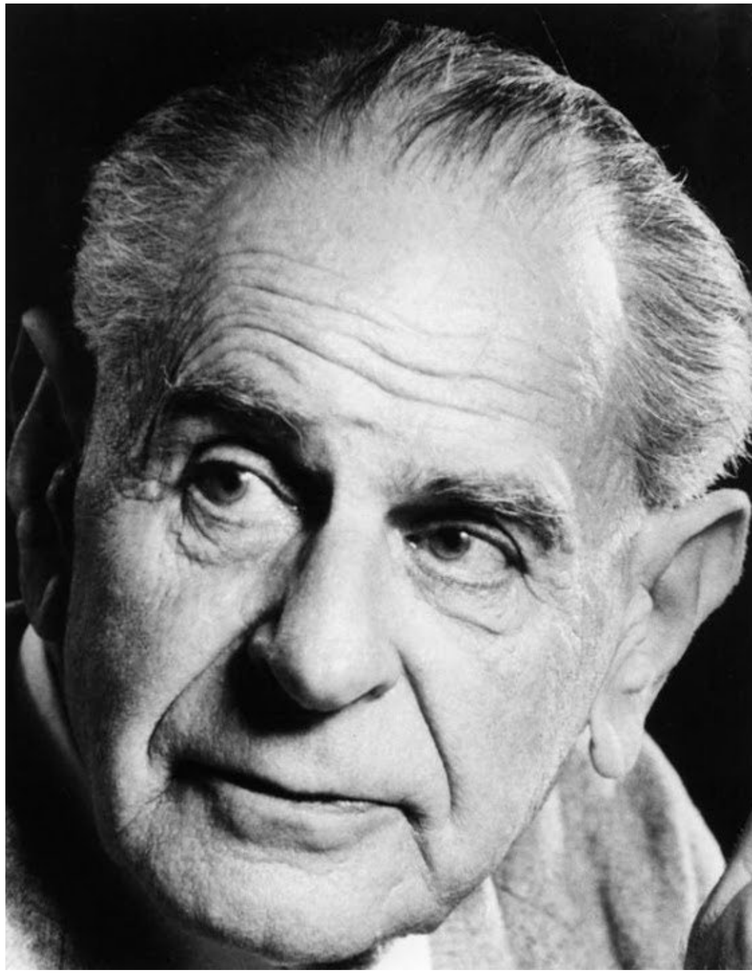


Karl Popper * 1902 in Vienna † 1994 in London

Scientific Enquiry as a Testing Problem

“I propose to replace [..] the question of the sources of our knowledge [e.g., how to identify the “best” scientific theory] by the entirely different question:
'How can we hope to detect and eliminate error?”'

“The proper answer to my question [..] is, I believe, ‘By critizing the theories or guesses of others and —if we train ourselves to do so— by critizing our own theories and guesses.”



Karl Popper * 1902 in Vienna † 1994 in London

How do we measure progress in automation?

How do we measure progress in automation?

(we demonstrate that each new technique solves the problem more effectively than the state-of-the-art [SOTA])

- How do we know if a technique solves the problem more effectively?

How do we measure progress in automation?

(we demonstrate that each new technique solves the problem more effectively than the state-of-the-art [SOTA])

- How do we know if a technique solves the problem more effectively?
  - We define a measure of success (e.g., max. code coverage for testing problem).

How do we measure progress in automation?

(we demonstrate that each new technique solves the problem more effectively than the state-of-the-art [SOTA])

- How do we know if a technique solves the problem more effectively?
  - We define a measure of success (e.g., max. code coverage for testing problem).
  - We choose a few representative problem instances (e.g., programs to test).

How do we measure progress in automation?

(we demonstrate that each new technique solves the problem more effectively than the state-of-the-art [SOTA])

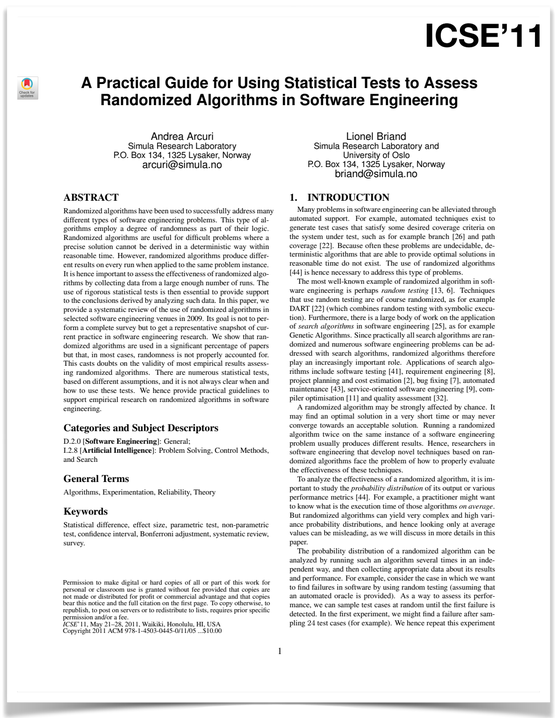
- How do we know if a technique solves the problem more effectively?
  - We define a measure of success (e.g., max. code coverage for testing problem).
  - We choose a few representative problem instances (e.g., programs to test).
  - Run a benchmarking framework to compare technique implementations.

How do we measure progress in automation?

(we demonstrate that each new technique solves the problem more effectively than the state-of-the-art [SOTA])

- How do we know if a technique solves the problem more effectively?
  - We define a measure of success (e.g., max. code coverage for testing problem).
  - We choose a few representative problem instances (e.g., programs to test).
  - Run a benchmarking framework to compare technique implementations.

ICSE'11



A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

Lionel Briand
Simula Research Laboratory and
University of Oslo
P.O. Box 134, 1325 Lysaker, Norway
briand@simula.no

ABSTRACT

Randomized algorithms have been used to successfully address many different types of software engineering problems. This type of algorithms employ a degree of randomness as part of their logic. Randomized algorithms are useful for difficult problems where a precise solution cannot be derived in a deterministic way within reasonable time. However, randomized algorithms produce different results on every run when applied to the same problem instance. It is hence important to assess the effectiveness of randomized algorithms by collecting data from a large enough number of runs. The use of rigorous statistical tests is then essential to provide support to the conclusions derived by analyzing such data. In this paper, we provide a systematic review of the use of randomized algorithms in selected software engineering venues in 2009. Its goal is not to perform a complete survey but to get a representative snapshot of current practice in software engineering research. We show that randomized algorithms are used in a significant percentage of papers but that, in most cases, randomness is not properly accounted for. This casts doubts on the validity of most empirical results assessing randomized algorithms. There are numerous statistical tests, based on different assumptions, and it is not always clear when and how to use these tests. We hence provide practical guidelines to support empirical research on randomized algorithms in software engineering.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Reliability, Theory

Keywords

Statistical difference, effect size, parametric test, non-parametric test, confidence interval, Bonferroni adjustment, systematic review, survey.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Many problems in software engineering can be alleviated through automated support. For example, automated techniques exist to generate test cases that satisfy some desired coverage criteria on the system under test, such as for example branch [26] and path coverage [22]. Because often these problems are undecidable, deterministic algorithms that are able to provide optimal solutions in reasonable time do not exist. The use of randomized algorithms [44] is hence necessary to address this type of problems.

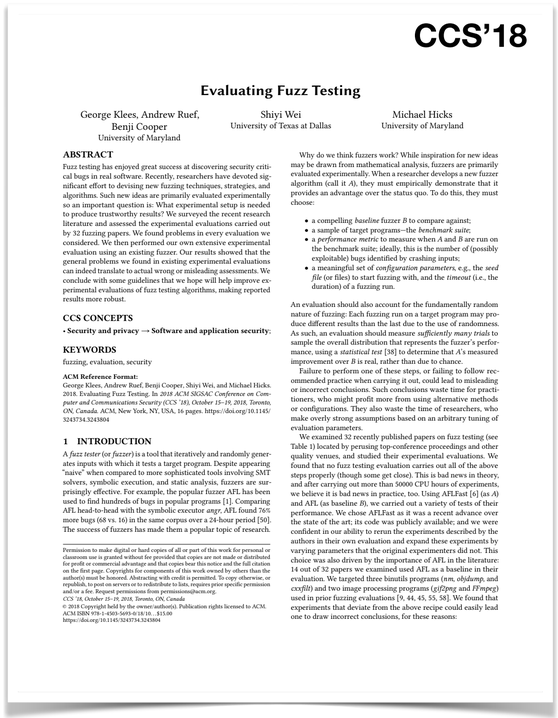
The most well-known example of randomized algorithm in software engineering is perhaps *random testing* [13, 6]. Techniques that use random testing are of course randomized, as for example DART [22] (which combines random testing with symbolic execution). Furthermore, there is a large body of work on the application of *search algorithms* in software engineering [25], as for example Genetic Algorithms. Since practically all search algorithms are randomized and numerous software engineering problems can be addressed with search algorithms, randomized algorithms therefore play an increasingly important role. Applications of search algorithms include software testing [41], requirement engineering [8], project planning and cost estimation [2], bug fixing [7], automated maintenance [43], service-oriented software engineering [9], compiler optimisation [11] and quality assessment [32].

A randomized algorithm may be strongly affected by chance. It may find an optimal solution in a very short time or may never converge towards an acceptable solution. Running a randomized algorithm twice on the same instance of a software engineering problem usually produces different results. Hence, researchers in software engineering that develop novel techniques based on randomized algorithms face the problem of how to properly evaluate the effectiveness of these techniques.

To analyze the effectiveness of a randomized algorithm, it is important to study the *probability distribution* of its output or various performance metrics [44]. For example, a practitioner might want to know what is the execution time of those algorithms *on average*. But randomized algorithms can yield very complex and high variance probability distributions, and hence looking only at average values can be misleading, as we will discuss in more details in this paper.

The probability distribution of a randomized algorithm can be analyzed by running such an algorithm several times in an independent way, and then collecting appropriate data about its results and performance. For example, consider the case in which we want to find failures in software by using random testing (assuming that an automated oracle is provided). As a way to assess its performance, we can sample test cases at random until the first failure is detected. In the first experiment, we might find a failure after sampling 24 test cases (for example). We hence repeat this experiment

CCS'18



Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Maryland

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

fuzzing, evaluation, security

ACM Reference Format:
George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243804

1 INTRODUCTION

A *fuzz tester* (or *fuzzer*) is a tool that iteratively and randomly generates inputs with which it tests a target program. Despite appearing "naive" when compared to more sophisticated tools involving SMT solvers, symbolic execution, and static analysis, fuzzers are surprisingly effective. For example, the popular fuzzer AFL has been used to find hundreds of bugs in popular programs [1]. Comparing AFL head-to-head with the symbolic executor *angr*, AFL found 76% more bugs (68 vs. 16) in the same corpus over a 24-hour period [50]. The success of fuzzers has made them a popular topic of research.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzer algorithm (call it *A*), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling *baseline* fuzzer *B* to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when *A* and *B* are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *seed file* (or files) to start fuzzing with, and the *timeout* (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random nature of fuzzing: Each fuzzing run on a target program may produce different results than the last due to the use of randomness. As such, an evaluation should measure *sufficiently many trials* to sample the overall distribution that represents the fuzzer's performance, using a statistical test [38] to determine that *A*'s measured improvement over *B* is real, rather than due to chance.

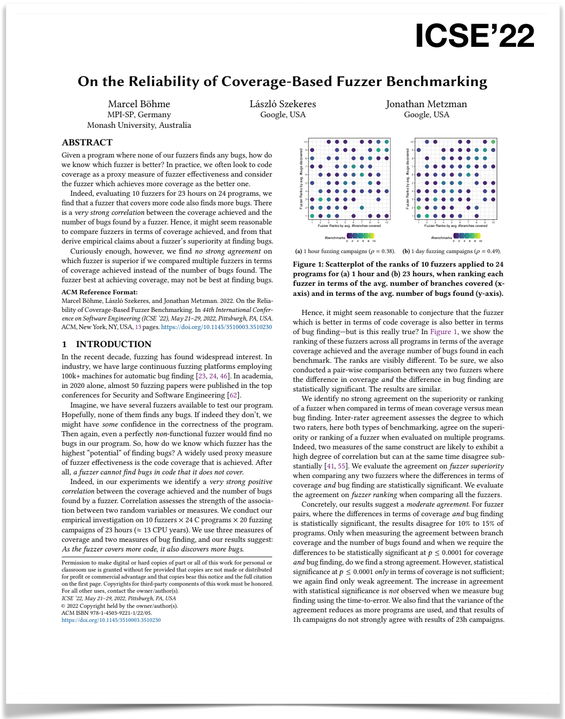
Failure to perform one of these steps, or failing to follow recommended practice when carrying it out, could lead to misleading or incorrect conclusions. Such conclusions waste time for practitioners, who might profit more from using alternative methods or configurations. They also waste the time of researchers, who make overly strong assumptions based on an arbitrary tuning of evaluation parameters.

We examined 32 recently published papers on fuzz testing (see Table 1) located by perusing top-conference proceedings and other quality venues, and studied their experimental evaluations. We found that no fuzz testing evaluation carries out all of the above steps properly (though some get close). This is bad news in theory, and after carrying out more than 50000 CPU hours of experiments, we believe it is bad news in practice, too. Using AFLFast [6] (as *A*) and AFL (as baseline *B*), we carried out a variety of tests of their performance. We chose AFLFast as it was a recent advance over the state of the art; its code was publicly available; and we were confident in our ability to rerun the experiments described by the authors in their own evaluation and expand these experiments by varying parameters that the original experimenters did not. This choice was also driven by the importance of AFL in the literature: 14 out of 32 papers we examined used AFL as a baseline in their evaluation. We targeted three binutils programs (*nm*, *objdump*, and *cxxfilt*) and two image processing programs (*gif2png* and *ffmpeg*) used in prior fuzzing evaluations [9, 44, 45, 55, 58]. We found that experiments that deviate from the above recipe could easily lead one to draw incorrect conclusions, for these reasons:

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243804

ICSE'22



On the Reliability of Coverage-Based Fuzzer Benchmarking

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

László Szekeres
Google, USA

Jonathan Metzman
Google, USA

ABSTRACT

Given a program where none of our fuzzers finds any bugs, how do we know which fuzzer is better? In practice, we often look to code coverage as a proxy measure of fuzzer effectiveness and consider the fuzzer which achieves more coverage as the better one.

Indeed, evaluating 10 fuzzers for 23 hours on 24 programs, we find that a fuzzer that covers more code also finds more bugs. There is a *very strong correlation* between the coverage achieved and the number of bugs found by a fuzzer. Hence, it might seem reasonable to compare fuzzers in terms of coverage achieved, and from that derive empirical claims about a fuzzer's superiority at finding bugs.

Curiously enough, however, we find *no strong agreement* on which fuzzer is superior if we compared multiple fuzzers in terms of coverage achieved instead of the number of bugs found. The fuzzer best at achieving coverage, may not be best at finding bugs.

ACM Reference Format:
Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510230

1 INTRODUCTION

In the recent decade, fuzzing has found widespread interest. In industry, we have large continuous fuzzing platforms employing 100k+ machines for automatic bug finding [23, 24, 46]. In academia, in 2020 alone, almost 50 fuzzing papers were published in the top conferences for Security and Software Engineering [62].

Imagine, we have several fuzzers available to test our program. Hopefully, none of them finds any bugs. If indeed they don't, we might have *some* confidence in the correctness of the program. Then again, even a perfectly *non*-functional fuzzer would find no bugs in our program. So, how do we know which fuzzer has the highest "potential" of finding bugs? A widely used proxy measure of fuzzer effectiveness is the code coverage that is achieved. After all, *a fuzzer cannot find bugs in code that it does not cover.*

Indeed, in our experiments we identify a *very strong positive correlation* between the coverage achieved and the number of bugs found by a fuzzer. Correlation assesses the strength of the association between two random variables or measures. We conduct our empirical investigation on 10 fuzzers × 24 C programs × 20 fuzzing campaigns of 23 hours (= 13 CPU years). We use three measures of coverage and two measures of bug finding, and our results suggest: *As the fuzzer covers more code, it also discovers more bugs.*

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510230

(a) 1 hour fuzzing campaigns ($p = 0.38$). **(b)** 1 day fuzzing campaigns ($p = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Hence, it might seem reasonable to conjecture that the fuzzer which is better in terms of code coverage is also better in terms of bug finding—but is this really true? In Figure 1, we show the ranking of these fuzzers across all programs in terms of the average coverage achieved and the average number of bugs found in each benchmark. The ranks are visibly different. To be sure, we also conducted a pair-wise comparison between any two fuzzers where the difference in coverage *and* the difference in bug finding are statistically significant. The results are similar.

We identify no strong agreement on the superiority or ranking of a fuzzer when compared in terms of mean coverage versus mean bug finding. Inter-rater agreement assesses the degree to which two raters, here both types of benchmarking, agree on the superiority or ranking of a fuzzer when evaluated on multiple programs. Indeed, two measures of the same construct are likely to exhibit a high degree of correlation but can at the same time disagree substantially [41, 55]. We evaluate the agreement on *fuzzer superiority* when comparing any two fuzzers where the differences in terms of coverage *and* bug finding are statistically significant. We evaluate the agreement on *fuzzer ranking* when comparing all the fuzzers.

Concretely, our results suggest a *moderate agreement*. For fuzzer pairs, where the differences in terms of coverage *and* bug finding is statistically significant, the results disagree for 10% to 15% of programs. Only when measuring the agreement between branch coverage and the number of bugs found and when we require the differences to be statistically significant at $p \leq 0.0001$ for coverage *and* bug finding, do we find a strong agreement. However, statistical significance at $p \leq 0.0001$ *only* in terms of coverage is not sufficient; we again find only weak agreement. The increase in agreement with statistical significance is *not* observed when we measure bug finding using the time-to-error. We also find that the variance of the agreement reduces as more programs are used, and that results of 1h campaigns do not strongly agree with results of 23h campaigns.

S&P'24

SoK: Prudent Evaluation Practices for Fuzzing

Moritz Schloegel[1], Nils Bars[1], Nico Schiller[1], Lukas Bernhard[1], Tobias Scharnowski[1]
Addison Crump[1], Arash Ale-Ebrahim[1], Nicolai Bissantz[2], Marius Muench[3], Thorsten Holz[1]

[1]*CISPA Helmholtz Center for Information Security, {first.lastname}@cispa.de*
[2]*Ruhr University Bochum, nicolai.bissantz@ruhr-uni-bochum.de*
[3]*University of Birmingham, m.muench@bham.ac.uk*

***Abstract*—Fuzzing has proven to be a highly effective approach to uncover software bugs over the past decade. After AFL popularized the groundbreaking concept of lightweight coverage feedback, the field of fuzzing has seen a vast amount of scientific work proposing new techniques, improving methodological aspects of existing strategies, or porting existing methods to new domains. All such work must demonstrate its merit by showing its applicability to a problem, measuring its performance, and often showing its superiority over existing works in a thorough, empirical evaluation. Yet, fuzzing is highly sensitive to its target, environment, and circumstances, e. g., randomness in the testing process. After all, relying on randomness is one of the core principles of fuzzing, governing many aspects of a fuzzer's behavior. Combined with the often highly difficult to control environment, the *reproducibility* of experiments is a crucial concern and requires a prudent evaluation setup. To address these threats to validity, several works, most notably *Evaluating Fuzz Testing* by Klees et al., have outlined how a carefully designed evaluation setup should be implemented, but it remains unknown to what extent their recommendations have been adopted in practice.**

In this work, we systematically analyze the evaluation of 150 fuzzing papers published at the top venues between 2018 and 2023. We study how existing guidelines are implemented and observe potential shortcomings and pitfalls. We find a surprising disregard of the existing guidelines regarding statistical tests and systematic errors in fuzzing evaluations. For example, when investigating reported bugs, we find that the search for vulnerabilities in real-world software leads to authors requesting and receiving CVEs of questionable quality. Extending our literature analysis to the practical domain, we attempt to reproduce claims of eight fuzzing papers. These case studies allow us to assess the practical reproducibility of fuzzing research and identify archetypal pitfalls in the evaluation design. Unfortunately, our reproduced results reveal several deficiencies in the studied papers, and we are unable to fully support and reproduce the respective claims. To help the field of fuzzing move toward a scientifically reproducible evaluation strategy, we propose updated guidelines for conducting a fuzzing evaluation that future work should follow.

1. Introduction

Fuzzing, a portmanteau of "fuzz testing", has gained much attention in recent years, and the method has proven to be highly successful in uncovering many types of faults in software systems. Companies such as Meta, Google, and Oracle have invested significant resources in this technology and use it to test their products. Large software projects such as web browsers or the Linux kernel incorporate fuzzing into their development cycle, and Google is running an extensive and continuous fuzzing campaign for more than 1, 200 open-source projects via OSS-Fuzz [62]. Beyond the wide acceptance in the industry, a large number of academic papers have proposed numerous improvements and novel techniques to enhance fuzzing further. More specifically, we found that, over the past six years, more than 280 papers on fuzzing have been published in the top computer security and software engineering venues.

A cornerstone of fuzzing research, and science in general, is that other researchers can critically assess the correctness of scientific results. To this end, the research results must be *reproducible*, meaning that another group should be able to obtain the same results using the same experimental setup, often by using a research artifact provided by the authors [8]. Reproducibility is paramount for other researchers to understand, trust, and build on the research results.

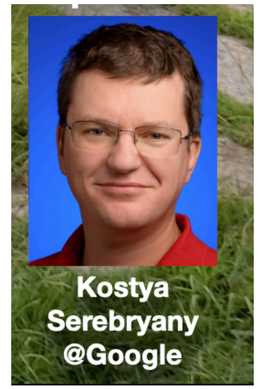
To enable high-quality research and provide a common foundation for evaluating fuzzing methods, several works describe how newly proposed fuzzing approaches should be evaluated. In 2018, the first and most influential paper describing a reproducible evaluation design was published by Klees et al. [88]. It describes guidelines to advise researchers on how fuzzing research should evaluate their respective contributions. For example, a crucial insight introduced by Klees et al. is the repetition of experiments to account for the inherent randomness of the fuzzing process. Although Klees et al. recommend "a sufficient number of trials" and use 30 trials in their own experiments, we found that in practice, this recommendation is interpreted as anything between three and 20 repetitions. Another guideline is to confirm the fuzzers' performance statistically; however, this makes little sense with few repetitions and is often skipped.

Benchmarks induce progress.

Benchmarks induce progress.

Benchmarks induce progress.

- Shonan Meeting: Benchmarking = Top-3 Research Challenge!
  - Google's commitment: Support of the community via FuzzBench.



Benchmarks induce progress.

- Shonan Meeting: Benchmarking = Top-3 Research Challenge!
  - Google's commitment: Support of the community via FuzzBench.
    - Highest standards of experimental design (20+ trials, 23hrs, 20+ programs).
    - 50+ fuzzers, 150+ experiments for 120+ papers (as of FSE'21), reproducible.

FSE'21

FuzzBench: An Open Fuzzer Benchmarking Platform and Service

Jonathan Metzman
Google, USA
metzman@google.com

László Szekeres
Google, USA
lszekeres@google.com

Laurent Simon
Google, USA
laurentsimon@google.com

Read Sprabery
Google, USA
sprabery@google.com

Abhishek Arya
Google, USA
aarya@google.com

ABSTRACT

Fuzzing is a key tool used to reduce bugs in production software. At Google, fuzzing has uncovered tens of thousands of bugs. Fuzzing is also a popular subject of academic research. In 2020 alone, over 120 papers were published on the topic of improving, developing, and evaluating fuzzers and fuzzing techniques. Yet, proper evaluation of fuzzing techniques remains elusive. The community has struggled to converge on methodology and standard tools for fuzzer evaluation.

To address this problem, we introduce FuzzBench as an open-source turnkey platform and free service for evaluating fuzzers. It aims to be easy to use, fast, reliable, and provides reproducible experiments. Since its release in March 2020, FuzzBench has been widely used both in industry and academia, carrying out more than 150 experiments for external users. It has been used by several published and in-the-work papers from academic groups, and has had real impact on the most widely used fuzzing tools in industry. The presented case studies suggest that FuzzBench is on its way to becoming a standard fuzzer benchmarking platform.

CCS CONCEPTS

• **Software and its engineering** → *Application specific development environments*; **Software testing and debugging**; • **Security and privacy** → **Software security engineering**; • **Mathematics of computing** → *Hypothesis testing and confidence interval computation*; • **General and reference** → **Evaluation**; **Experimentation**.

KEYWORDS

fuzzing, fuzz testing, benchmarking, testing, software security

ACM Reference Format:
Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473932

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473932

1 INTRODUCTION

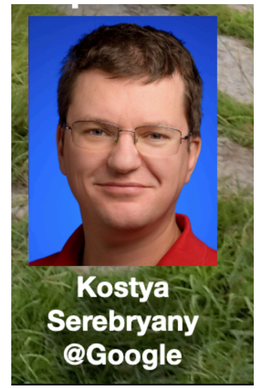
Fuzzing has attracted the attention of both industry and academia because it is effective at finding bugs in real-world software, not just in experiments. Today, fuzzing has seen high adoption among developers [34] and is used to find bugs in widely used production software [26, 27, 40, 42]. At Google we have found tens of thousands of bugs [1] with fuzzers like AFL [45], libFuzzer [37] and Honggfuzz [43]. Academic research on fuzzing has driven many improvements since the inception of coverage-guided fuzzing [45] – Google Scholar reports several thousand published papers since 2014 [28].

While fuzzing efforts have been successful in improving software quality, proper evaluation of fuzzing techniques is still a challenge. There is no consensus on which tools and techniques are effective and generalize well for fuzzer comparison. This is in part due to the lack of standard benchmarking tools, metrics, and representative program datasets, all of which have hampered reproducibility [48].

Klees et al. [31] were the first to study the current state of fuzzing evaluations. They analyzed 32 fuzzing research papers and found that none provided enough "evidence to justify general claims of effectiveness". More specifically, some papers do not use a large and diverse set of real-world benchmarks, have too few trials, use short trials, or lack statistical tests. Furthermore, it is hard to cross-compare between all papers as they typically use different evaluation setup and configuration (e.g., how experiments are run and measured), different subjects (benchmark programs) or even different coverage metrics [41].

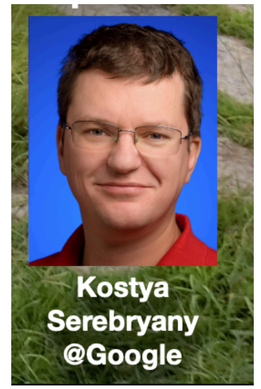
Another common challenge is that sound fuzzer evaluation has a high cost, both in researcher time and computational resources. A typical evaluation compares a large number of tools on a large number of subjects (benchmark programs). Setting up all these tools and subjects and making sure that each tool-subject pair works together (i.e., compiles, runs) takes significant effort. Some researchers we talked to described spending several months working on evaluation. A sound evaluation also needs massive computation time (on the order of CPU-years) and resources, as each tool-subject pair needs to run multiple times for statistical significance. In practice, it can take up to ~ 11 CPU-years to run a well-conducted experiment (e.g., 24 hours × 20 trials × 10 fuzzers × 20 subjects). On Google Cloud, this experiment could cost over $2,000. Considering the repeated evaluations necessary during the development of a fuzzing tool, research can require CPU-centuries and tens of thousands of dollars.

FuzzBench aims to alleviate these problems by providing an open-source fuzzer benchmarking service. We designed it following



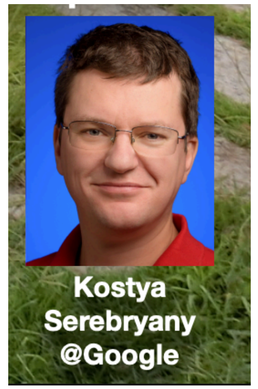
Benchmarks induce progress.

- Shonan Meeting: Benchmarking = Top-3 Research Challenge!
  - Google's commitment: Support of the community via FuzzBench.
    - Highest standards of experimental design (20+ trials, 23hrs, 20+ programs).
    - 50+ fuzzers, 150+ experiments for 120+ papers (as of FSE'21), reproducible.
    - Used in SBFT Fuzzing competitions (since 2023).



Benchmarks induce progress.

- Shonan Meeting: Benchmarking = Top-3 Research Challenge!
  - Google's commitment: Support of the community via FuzzBench.
    - Highest standards of experimental design (20+ trials, 23hrs, 20+ programs).
    - 50+ fuzzers, 150+ experiments for 120+ papers (as of FSE'21), reproducible.
    - Used in SBFT Fuzzing competitions (since 2023).
    - Enabled major advances in industrial fuzzers:
      AFL++, LibAFL, LibFuzzer, Honggfuzz, and Centipede.

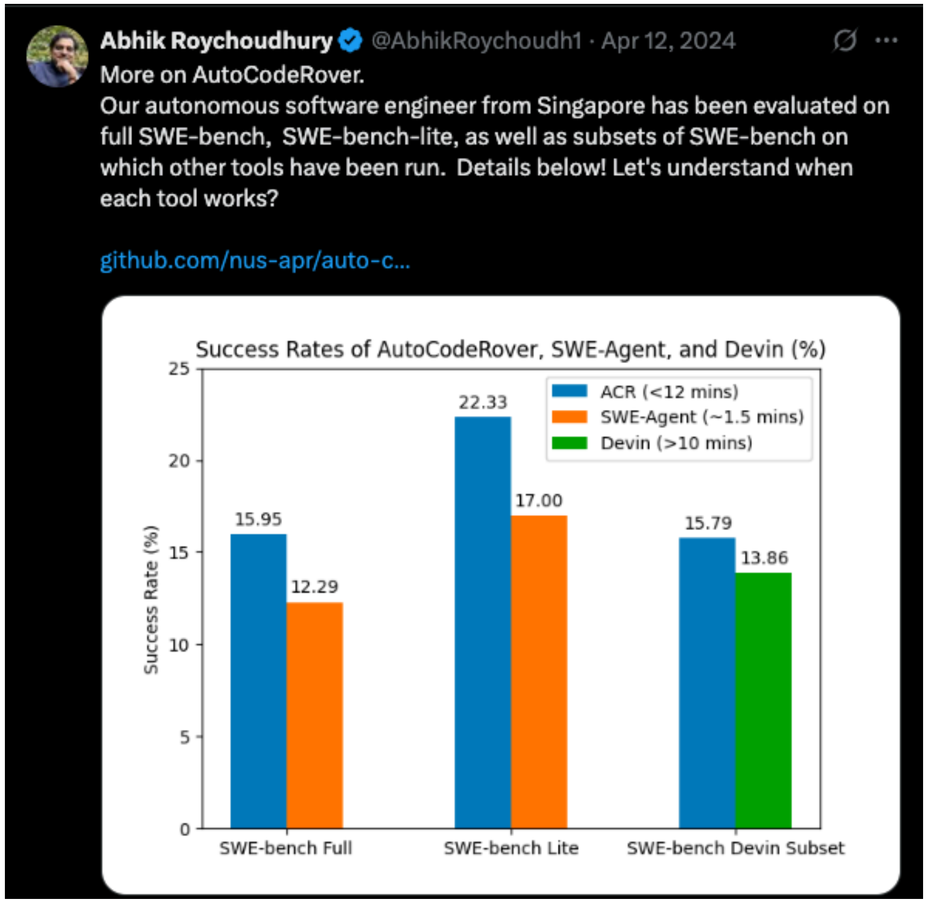


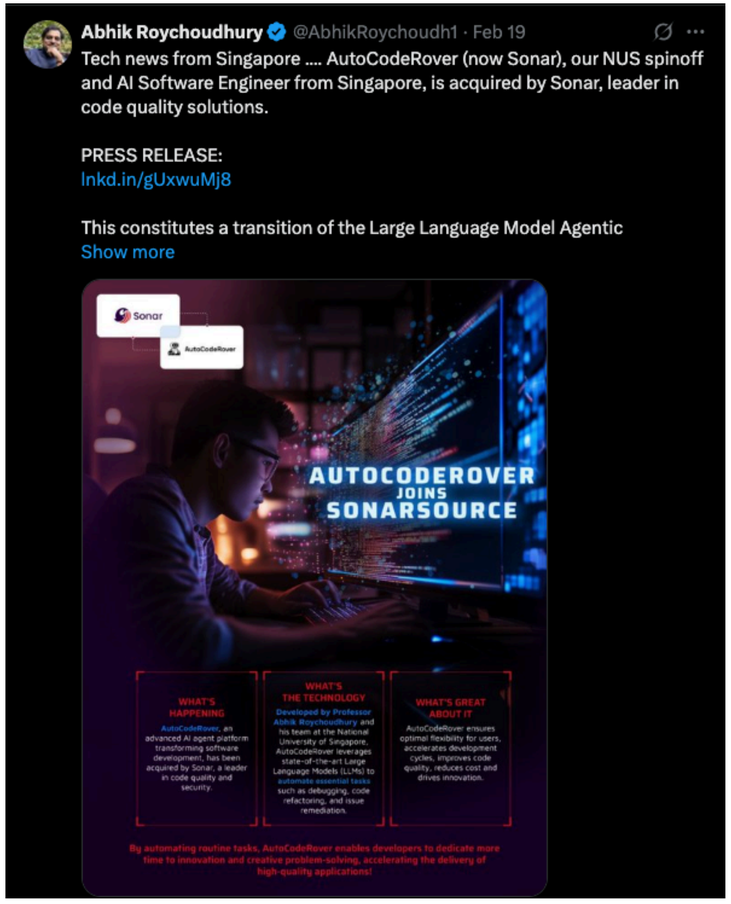
Benchmarks induce progress.

- Shonan Meeting: Benchmarking = Top-3 Research Challenge!
  - Google's commitment: Support of the community via FuzzBench.
    - Highest standards of experimental design (20+ trials, 23hrs, 20+ programs).
    - 50+ fuzzers, 150+ experiments for 120+ papers (as of FSE'21), reproducible.
    - Used in SBFT Fuzzing competitions (since 2023).
    - Enabled major advances in industrial fuzzers:
      AFL++, LibAFL, LibFuzzer, Honggfuzz, and Centipede.
- Today, there many other fuzzer benchmark frameworks.
  - Magma: https://github.com/HexHive/magma
  - Fuzztastic: https://github.com/tum-i4/fuzztastic
  - UniBench: https://github.com/unifuzz/unibench
  - ProFuzzBench: https://github.com/profuzzbench/profuzzbench

Benchmarks induce progress.

- Benchmarking to measure progress in all of automation.
  - Automated Software Engineering: SWE-Bench, Defects4J, CoREBench.



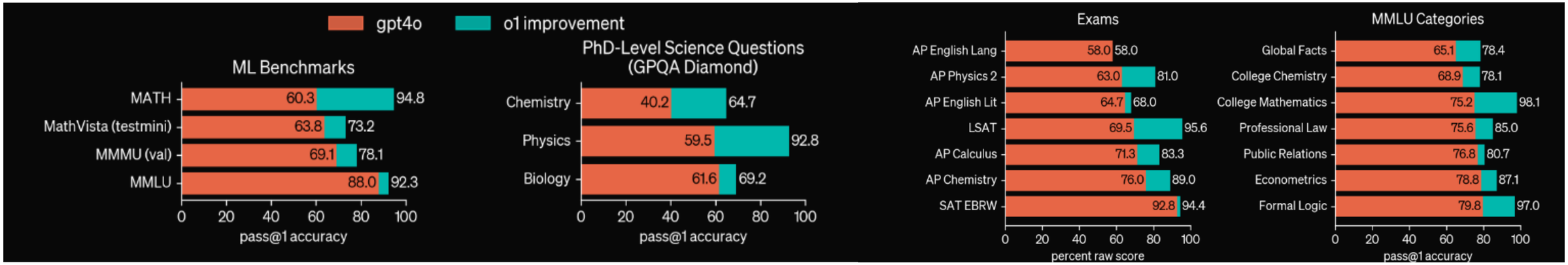


Benchmarks induce progress.

- Benchmarking to measure progress in all of automation.
  - Automated Software Engineering: SWE-Bench, Defects4J, CoREBench.
  - Automated Cybersecurity: DARPA CGC, AIxCC (8.5 *million* USD in prizes)

Benchmarks induce progress.

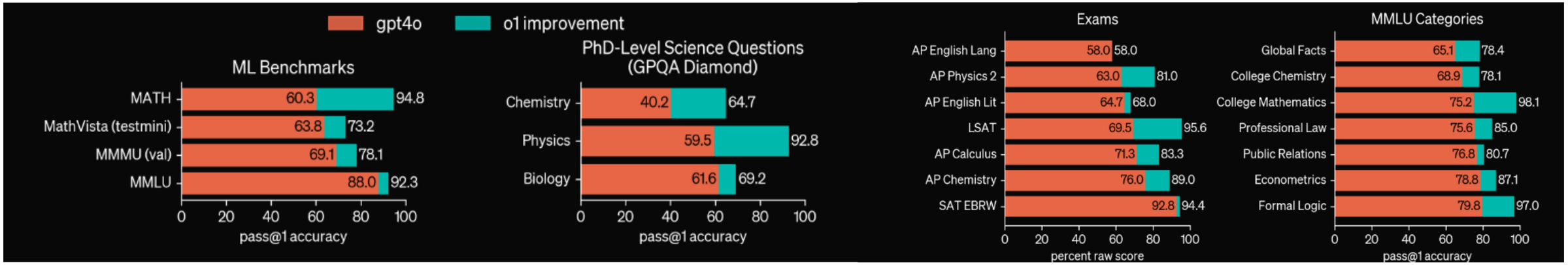
- Benchmarking to measure progress in all of automation.
  - Automated Software Engineering: SWE-Bench, Defects4J, CoREBench.
  - Automated Cybersecurity: DARPA CGC, AIxCC (8.5 *million* USD in prizes)
  - Machine Learning / Artificial Intelligence:
    - ARC Challenge (1+ million USD in prizes).
    - Most ML/AI conferences have a track to announce new benchmarks.
    - Every announcement of a new LLM comes with results on popular benchmarks.



There are limits to benchmarking.

- Does **100%** on all currently known ML benchmarks mean AGI?
- Does going from **92%** to **95%** mean substantial progress?

Every announcement of a new LLM comes with results on popular benchmarks.

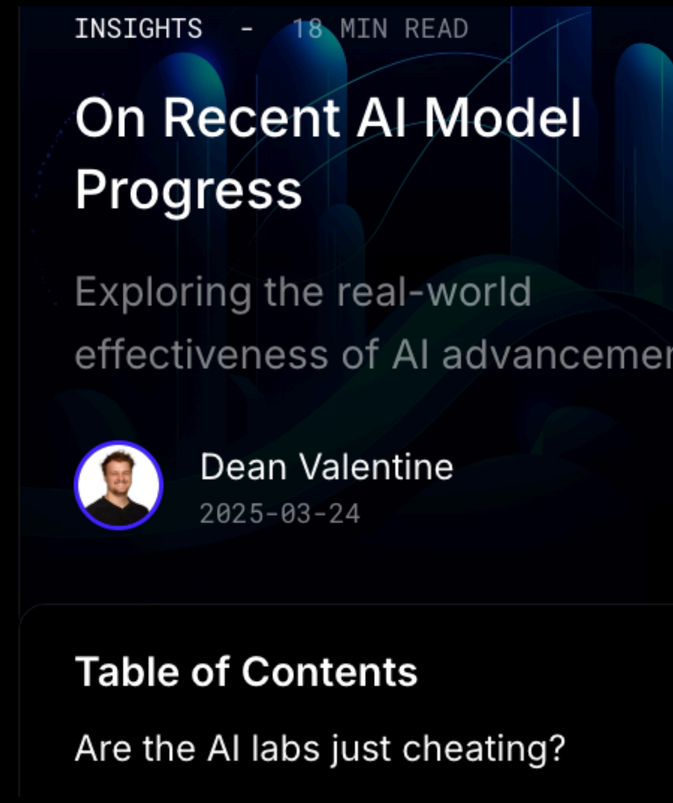


There are limits to benchmarking.

- Does **100%** on all currently known ML benchmarks mean AGI?
- Does going from **92%** to **95%** mean substantial progress?

But in recent months I've spoken to other YC founders doing AI application startups and most of them have had the same anecdotal experiences: 1. o99-pro-ultra announced, 2. Benchmarks look good, 3. Evaluated performance mediocre. This is despite the fact that we work in different industries, on different problem sets. Sometimes the founder will apply a cope to the narrative ("We just don't have any PhD level questions to ask"), but the narrative is there.

I have read the studies. I have seen the numbers. Maybe LLMs are becoming more fun to talk to, maybe they're performing better on controlled exams. But I would nevertheless like to submit, based off of internal benchmarks, and my own and colleagues' perceptions using these models, that whatever gains these companies are reporting to the public, they are not reflective of economic usefulness or generality. They are not reflective of my Lived Experience or the Lived Experience of my customers. In terms of being able to perform entirely new tasks, or larger proportions of users' intellectual labor, I don't think they have improved much since August.



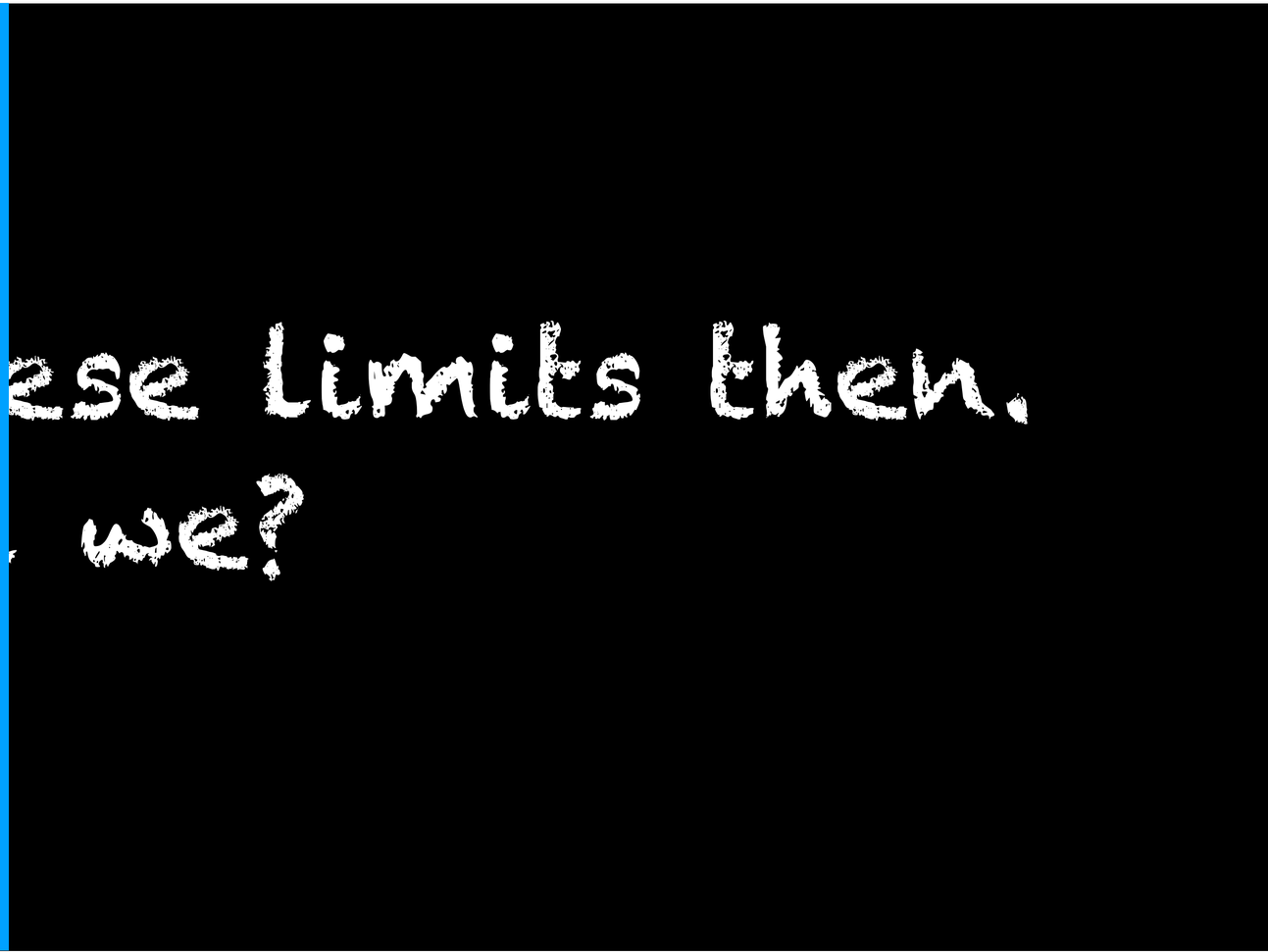
There are limits to benchmarking.

Let's explore these limits then.
Shall we?

There are limits to benchmarking.

Section I

Automated
Software Testing



There are limits to benchmarking.

Section I

Automated
Software Testing

Section II

Automated
Vulnerability Discovery

There are limits to benchmarking.

Section I

Automated
Software Testing

Benchmarking Fuzzers

Suppose none of our fuzzers
finds any bugs in our program.

How do we know which fuzzer is better?

Benchmarking Fuzzers Using Coverage

Suppose none of our fuzzers
finds any bugs in our program.

How do we know which fuzzer is better?

We measure code coverage!

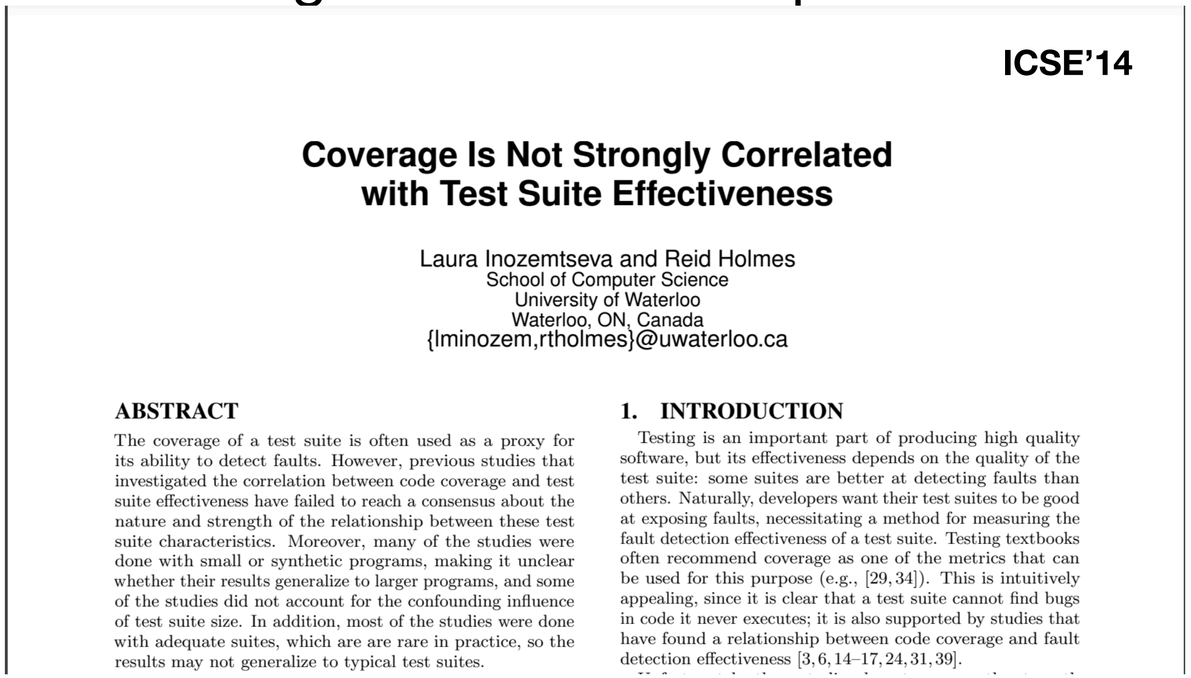
Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

We measure code coverage!

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?



ICSE'14

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

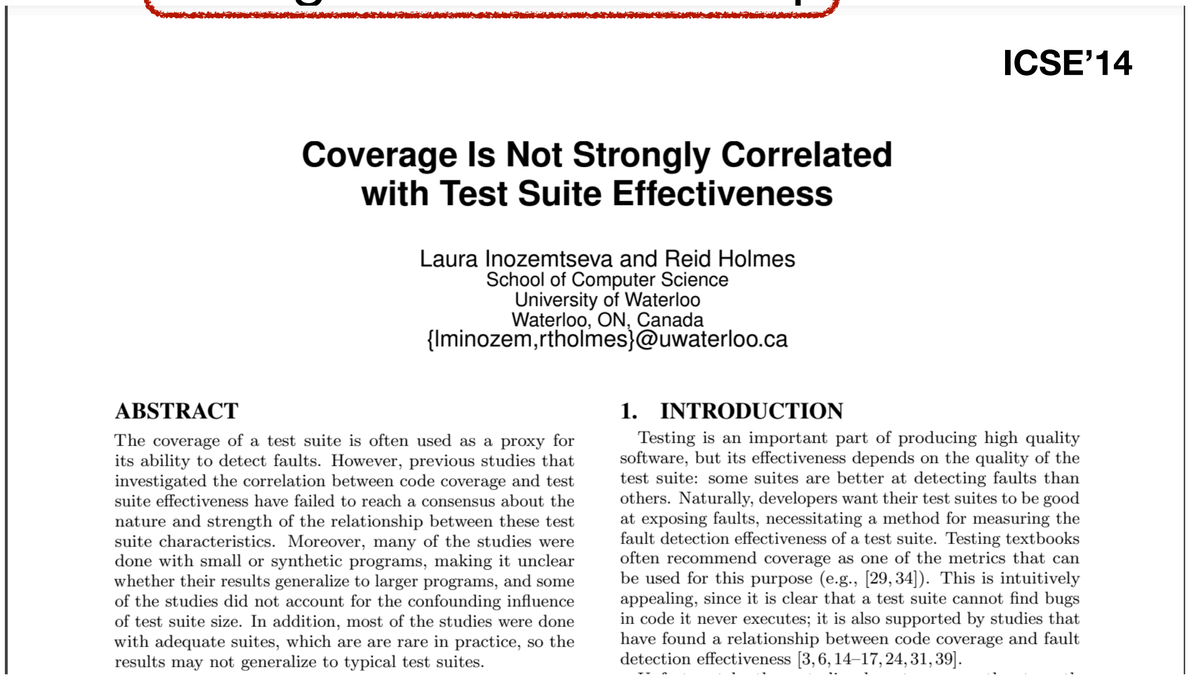
1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".



ICSE'14

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

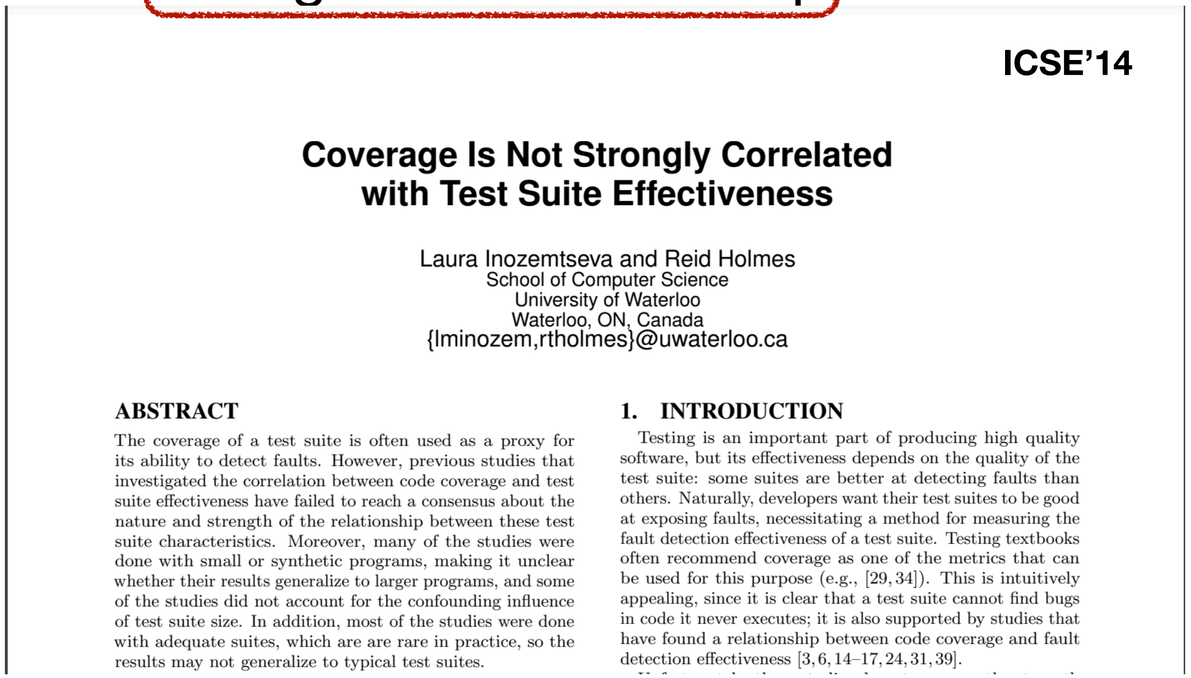
1. INTRODUCTION

Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".



ICSE'14

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

ABSTRACT

The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size. In addition, most of the studies were done with adequate suites, which are are rare in practice, so the results may not generalize to typical test suites.

1. INTRODUCTION

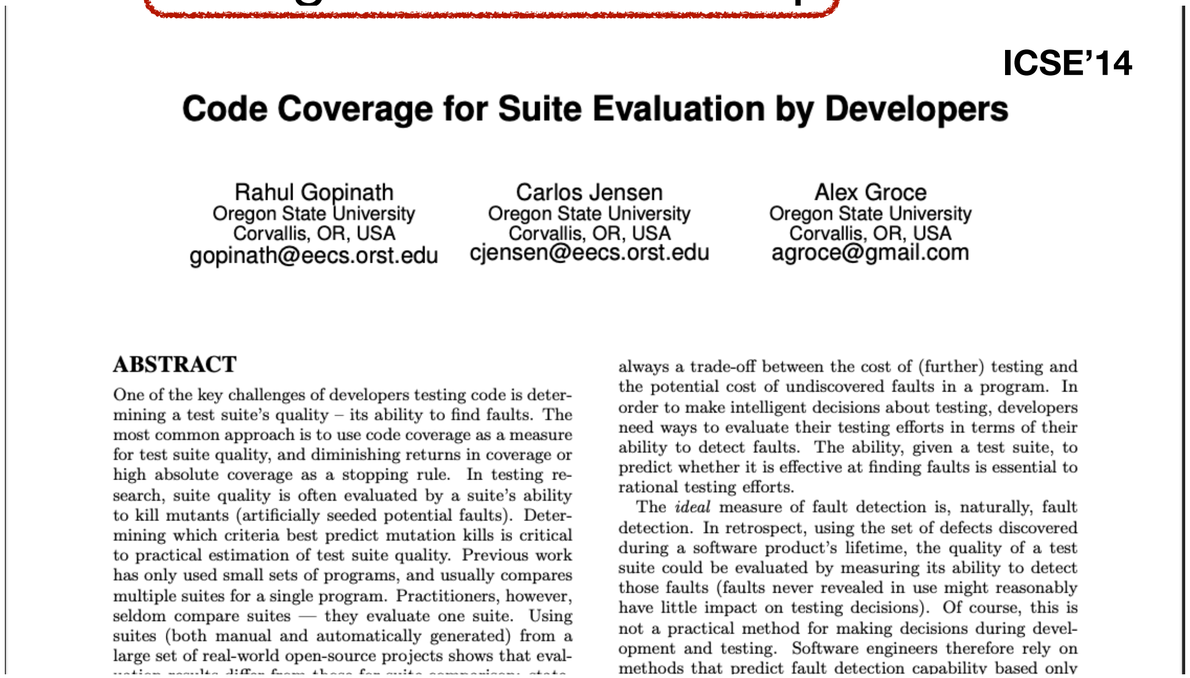
Testing is an important part of producing high quality software, but its effectiveness depends on the quality of the test suite: some suites are better at detecting faults than others. Naturally, developers want their test suites to be good at exposing faults, necessitating a method for measuring the fault detection effectiveness of a test suite. Testing textbooks often recommend coverage as one of the metrics that can be used for this purpose (e.g., [29, 34]). This is intuitively appealing, since it is clear that a test suite cannot find bugs in code it never executes; it is also supported by studies that have found a relationship between code coverage and fault detection effectiveness [3, 6, 14–17, 24, 31, 39].

- **Observation**: Test suites with more coverage find more bugs only because they are bigger.

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".



ICSE'14

Code Coverage for Suite Evaluation by Developers

Rahul Gopinath
Oregon State University
Corvallis, OR, USA
gopinath@eecs.orst.edu

Carlos Jensen
Oregon State University
Corvallis, OR, USA
cjensen@eecs.orst.edu

Alex Groce
Oregon State University
Corvallis, OR, USA
agroce@gmail.com

ABSTRACT

One of the key challenges of developers testing code is determining a test suite's quality – its ability to find faults. The most common approach is to use code coverage as a measure for test suite quality, and diminishing returns in coverage or high absolute coverage as a stopping rule. In testing research, suite quality is often evaluated by a suite's ability to kill mutants (artificially seeded potential faults). Determining which criteria best predict mutation kills is critical to practical estimation of test suite quality. Previous work has only used small sets of programs, and usually compares multiple suites for a single program. Practitioners, however, seldom compare suites — they evaluate one suite. Using suites (both manual and automatically generated) from a large set of real-world open-source projects shows that eval-

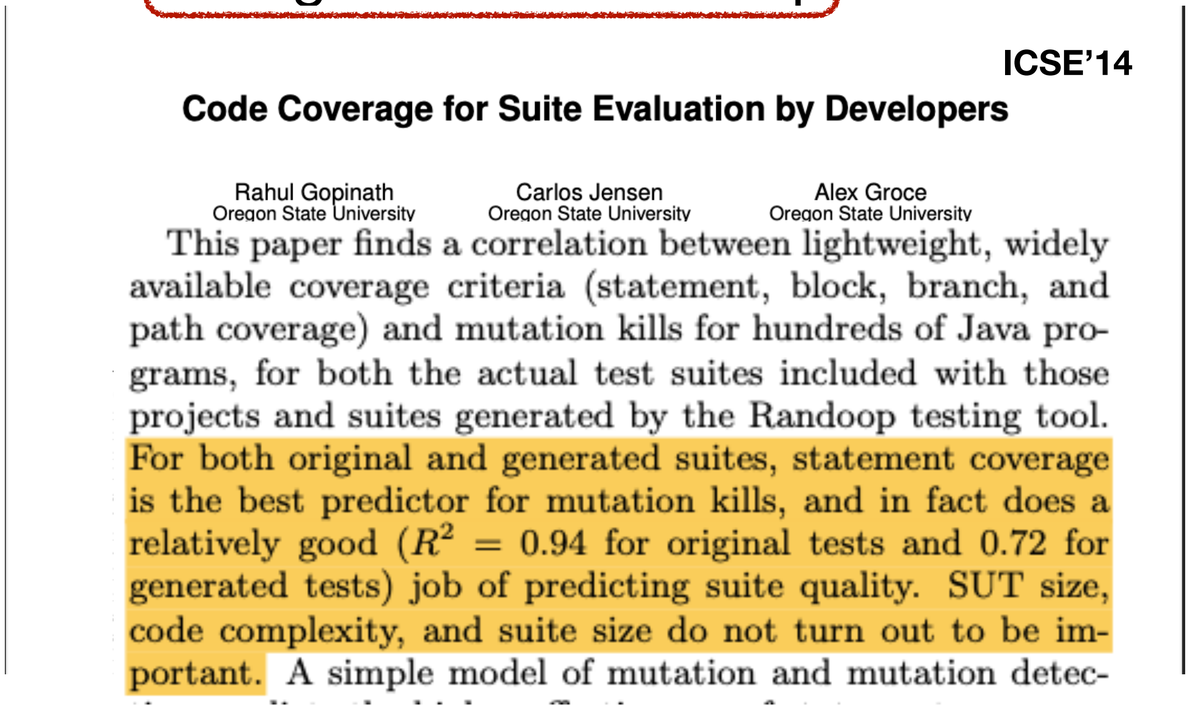
always a trade-off between the cost of (further) testing and the potential cost of undiscovered faults in a program. In order to make intelligent decisions about testing, developers need ways to evaluate their testing efforts in terms of their ability to detect faults. The ability, given a test suite, to predict whether it is effective at finding faults is essential to rational testing efforts.

The *ideal* measure of fault detection is, naturally, fault detection. In retrospect, using the set of defects discovered during a software product's lifetime, the quality of a test suite could be evaluated by measuring its ability to detect those faults (faults never revealed in use might reasonably have little impact on testing decisions). Of course, this is not a practical method for making decisions during development and testing. Software engineers therefore rely on methods that predict fault detection capability based only

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".



ICSE'14

Code Coverage for Suite Evaluation by Developers

Rahul Gopinath
Oregon State University

Carlos Jensen
Oregon State University

Alex Groce
Oregon State University

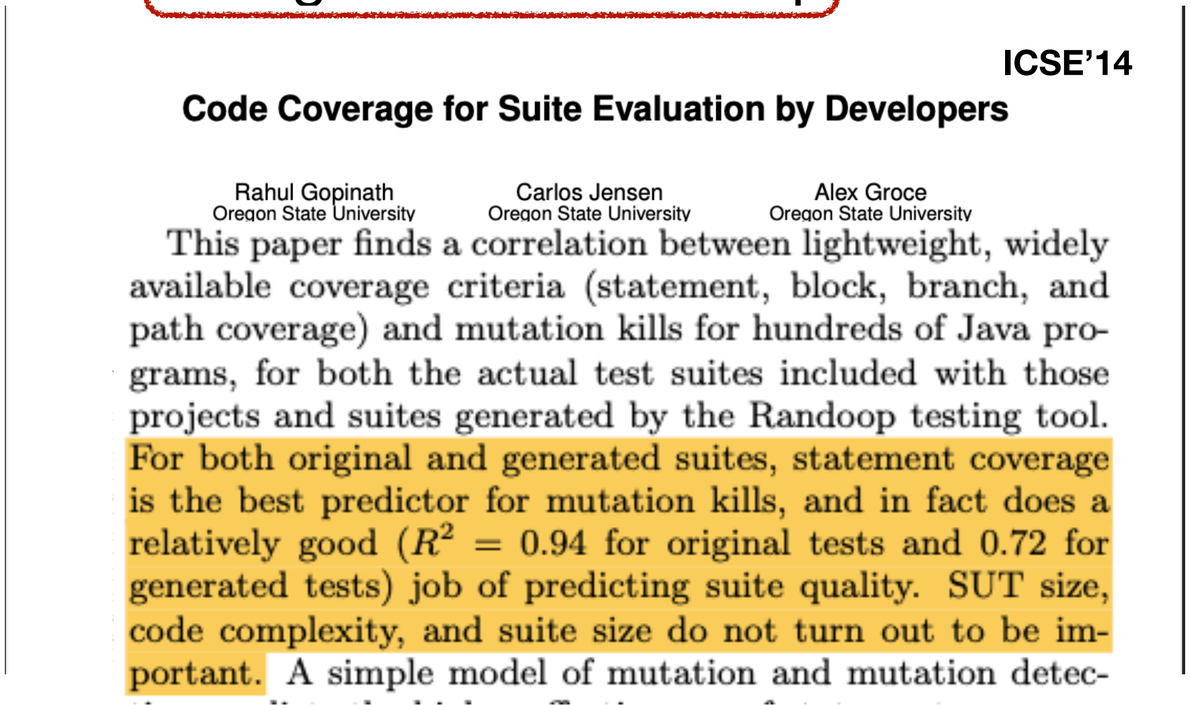
This paper finds a correlation between lightweight, widely available coverage criteria (statement, block, branch, and path coverage) and mutation kills for hundreds of Java programs, for both the actual test suites included with those projects and suites generated by the Randoop testing tool. For both original and generated suites, statement coverage is the best predictor for mutation kills, and in fact does a relatively good ($R^2 = 0.94$ for original tests and 0.72 for generated tests) job of predicting suite quality. SUT size, code complexity, and suite size do not turn out to be important. A simple model of mutation and mutation detec-

- **Observation**: Test suites with more coverage find more bugs **irrespective** of whether they are bigger.

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".



ICSE'14

Code Coverage for Suite Evaluation by Developers

Rahul Gopinath
Oregon State University

Carlos Jensen
Oregon State University

Alex Groce
Oregon State University

This paper finds a correlation between lightweight, widely available coverage criteria (statement, block, branch, and path coverage) and mutation kills for hundreds of Java programs, for both the actual test suites included with those projects and suites generated by the Randoop testing tool. For both original and generated suites, statement coverage is the best predictor for mutation kills, and in fact does a relatively good ($R^2 = 0.94$ for original tests and 0.72 for generated tests) job of predicting suite quality. SUT size, code complexity, and suite size do not turn out to be important. A simple model of mutation and mutation detec-

- **Observation**: Test suites with more coverage find more bugs **irrespective** of whether they are bigger.

This is called "contradiction".

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship between coverage and bug finding?

This is called "correlation".

ASE'20

Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size

Yiqun T. Chen, University of Washington, Seattle, WA, USA
Rahul Gopinath, CISPA Helmholtz-Zentrum, Saarbrücken, Germany
Anita Tadakamalla, George Mason University, Fairfax, VA, USA
Michael D. Ernst, University of Washington, Seattle, WA, USA
Reid Holmes, University of British Columbia, Vancouver, BC, Canada
Gordon Fraser, University of Passau, Passau, Germany
Paul Ammann, George Mason University, Fairfax, VA, USA
René Just, University of Washington, Seattle, WA, USA

ABSTRACT

The research community has long recognized a complex interrelationship between fault detection, test adequacy criteria, and test set size. However, there is substantial confusion about whether and how to experimentally control for test set size when assessing how well an adequacy criterion is correlated with fault detection and when comparing test adequacy criteria. Resolving the confusion, this paper makes the following contributions: (1) A review of contradictory analyses of the relationships between fault detection, test adequacy criteria, and test set size. Specifically, this paper addresses the supposed contradiction of prior work and explains why test set size is neither a confounding variable, as previously suggested, nor an independent variable that should be experimentally

Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416667

1 INTRODUCTION

The software engineering research community has long recognized a complex interrelationship between three variables:

- *fault detection* (the degree to which a test set detects real faults),
- *test set adequacy* (the degree to which a test set satisfies a set of test goals, such as statements, branches, or mutants), and
- *test set size* (the cardinality of a test set).

al. [44]. These papers report contradictory conclusions about:

- whether and how test set size should be experimentally controlled when assessing the correlation between test set adequacy and fault detection, and
- whether the correlation between test set adequacy and fault detection is significant and strong.

While some previous work notes these conflicts without providing resolutions, of greater concern is the fact that many other papers simply cite the aforementioned papers without noting the contradictions. (As of August 2020, Google Scholar reports over 800 citations to these four papers.)

Benchmarking Fuzzers Using Coverage

- Key Idea:
  - You cannot find bugs in code that is not covered.
- Question:
  - How strong is the relationship?

This is called "correlation".

ASE'20

Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size

Yiqun T. Chen, University of Washington, Seattle, WA, USA
Rahul Gopinath, CISPA Helmholtz-Zentrum, Saarbrücken, Germany
Anita Tadakamalla, George Mason University, Fairfax, VA, USA
Michael D. Ernst, University of Washington, Seattle, WA, USA
Reid Holmes, University of British Columbia, Vancouver, BC, Canada
Gordon Fraser, University of Passau, Passau, Germany
Paul Ammann, George Mason University, Fairfax, VA, USA
René Just, University of Washington, Seattle, WA, USA

ABSTRACT

The research community has long recognized a complex interrelationship between fault detection, test adequacy criteria, and test set size. However, there is substantial confusion about whether and how to experimentally control for test set size when assessing how well an adequacy criterion is correlated with fault detection and when comparing test adequacy criteria. Resolving the confusion, this paper makes the following contributions: (1) A review of contradictory analyses of the relationships between fault detection, test adequacy criteria, and test set size. Specifically, this paper addresses the supposed contradiction of prior work and explains why test set size is neither a confounding variable, as previously sug-

Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416667

1 INTRODUCTION

The software engineering research community has long recognized a complex interrelationship between three variables:

- *fault detection* (the degree to which a test set detects real faults),
- *test set adequacy* (the degree to which a test set satisfies a set of test goals, such as statements, branches, or mutants), and
- *test set size* (the cardinality of a test set).

8 CONCLUSIONS

This paper addresses and resolves the contradictions in prior work that studied the interrelationship between fault detection, test adequacy criteria, and test set size. It explains why test set size is an unrealistic test objective and neither a confounding variable nor an independent variable that should be experimentally manipulated. Furthermore, it explains the conceptual and statistical issues that arise when controlling for test set size via random selection and stratification, concluding that the random-selection methodology is flawed.

Additionally, this paper proposes (1) a methodology for comparing test adequacy criteria on a fair basis, accounting for test set size without direct, unrealistic manipulation, and (2) probabilistic coupling, a methodology for approximating the fault-detection probability of adequate test sets. Using the proposed methodology, this paper concludes that adequacy-based test selection is superior to random selection and that mutation-based test selection is most effective when employed after coverage has exhausted its usefulness.

Correlation: Very strong

- Our experiments confirm a very strong correlation for fuzzer-generated test suites!

- As a fuzzer covers more code, it also finds more bugs.

Correlation: Very strong

- Our experiments confirm a very strong correlation for fuzzer-generated test suites!
- As a fuzzer covers more code, it also finds more bugs.

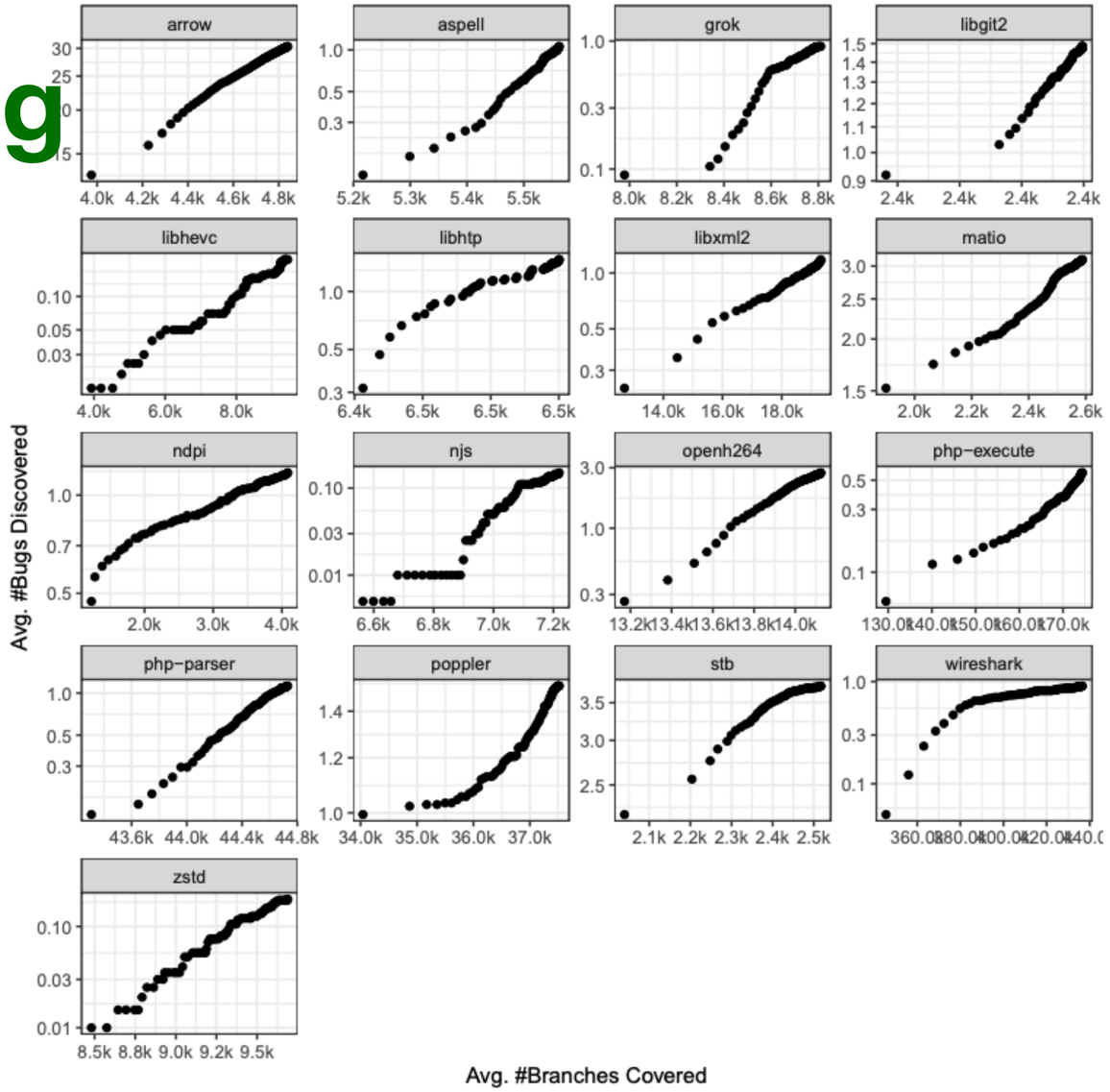


Figure 6: Scatter plot of the mean number of bugs found (on the log-scale) as the mean number of covered branches increases in the average fuzzing campaign for a benchmark.

Correlation: Very strong

- Our experiments confirm a very strong correlation for fuzzer-generated test suites!
- As a fuzzer covers more code, it also finds more bugs.

	#Branches	#Paths	#Edges
arrow	0.999269	0.999276	0.999277
matio	0.990898	0.990896	0.990892
ndpi	0.888853	0.888625	0.888602
njs	0.918627	0.918636	0.918627
openh264	0.969526	0.969552	0.969522
poppler	0.949209	0.949217	0.949210
wireshark	0.888212	0.888212	0.888212
aspell	0.988724	0.988689	0.988703
grok	0.880887	0.880876	0.880710
libgit2	0.605309	0.600231	0.602031
libhevc	0.959148	0.959149	0.959147
libhtp	0.974873	0.965578	0.975135
libxml2	0.932176	0.932191	0.932172
php-execute	0.834285	0.834286	0.834285
php-parser	0.989402	0.989377	0.989400
stb	0.951317	0.951294	0.951250
zstd	0.830236	0.830244	0.830233
Average	*0.914762*	*0.913902*	*0.914553*

Figure 7: Average correlation (ρ) between coverage and #bugs found for all programs where at least one bug was found.

Correlation: Very strong

- Our experiments confirm a very strong correlation for fuzzer-generated test suites!
- As a fuzzer covers more code, it also finds more bugs.

Spearman's ρ	Interpretation
0.00 - 0.09	Neglible correlation
0.10 - 0.39	Weak correlation
0.40 - 0.69	Moderate correlation
0.70 - 0.89	Strong correlation
0.90 - 1.00	Very strong correlation

	#Branches	#Paths	#Edges
arrow	0.999269	0.999276	0.999277
matio	0.990898	0.990896	0.990892
ndpi	0.888853	0.888625	0.888602
njs	0.918627	0.918636	0.918627
openh264	0.969526	0.969552	0.969522
poppler	0.949209	0.949217	0.949210
wireshark	0.888212	0.888212	0.888212
aspell	0.988724	0.988689	0.988703
grok	0.880887	0.880876	0.880710
libgit2	0.605309	0.600231	0.602031
libhevc	0.959148	0.959149	0.959147
libhtp	0.974873	0.965578	0.975135
libxml2	0.932176	0.932191	0.932172
php-execute	0.834285	0.834286	0.834285
php-parser	0.989402	0.989377	0.989400
stb	0.951317	0.951294	0.951250
zstd	0.830236	0.830244	0.830233
Average	*0.914762*	*0.913902*	*0.914553*

Figure 7: Average correlation (ρ) between coverage and #bugs found for all programs where at least one bug was found.

Quick Detour

Correlation: Very strong

TOSEM'18

STADS: Software Testing as Species Discovery

Spatial and Temporal Extrapolation from Tested Program Behaviors

MARCEL BÖHME*, National University of Singapore *and* Monash University, Australia

FSE'23

Statistical Reachability Analysis

Seongmin Lee
Max Planck Institute for Security and Privacy
Bochum, Germany
seongmin.lee@mpi-sp.org

Marcel Böhme
Max Planck Institute for Security and Privacy
Bochum, Germany
marcel.boehme@acm.org

ICSE'19

2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

Assurance in Software Testing: A Roadmap

Marcel Böhme
Monash University
marcel.boehme@acm.com

ICSE'24

Extrapolating Coverage Rate in Greybox Fuzzing

Danushka Liyanage*
Monash University
Australia

Seongmin Lee*
MPI-SP
Germany

Chakkrit Tantithamthavorn
Monash University
Australia

Marcel Böhme
MPI-SP
Germany

FSE'21

Estimating Residual Risk in Greybox Fuzzing

Marcel Böhme
Monash University, Australia
MPI-SP, Germany

Danushka Liyanage
Monash University
Australia

Valentin Wüstholz
ConsenSys
Germany

ABSTRACT

For any errorless fuzzing campaign, no matter how long, there is always some residual risk that a software error would be discovered if only the campaign was run for just a bit longer. Recently, greybox fuzzing tools have found widespread adoption. Yet, practitioners can only guess when the residual risk of a greybox fuzzing campaign falls below a specific, maximum allowable threshold.

In this paper, we explain why residual risk cannot be directly estimated for greybox campaigns, argue that the discovery probability (i.e., the probability that the next generated input increases code coverage) provides an excellent upper bound, and explore sound statistical methods to estimate the discovery probability in an ongoing greybox campaign. We find that estimators for blackbox fuzzing systematically and substantially *under*-estimate the true risk. An engineer—who stops the campaign when the estimators purport a risk below the maximum allowable risk—is vastly misled. She might need execute a campaign that is orders of magnitude longer to achieve the allowable risk. Hence, the *key challenge* we address in this paper is *adaptive bias*: The probability to discover a specific error actually increases over time. We provide the first probabilistic analysis of adaptive bias, and introduce two novel classes of estimators that tackle adaptive bias. With our estimators, the engineer can decide with confidence when to abort the campaign.

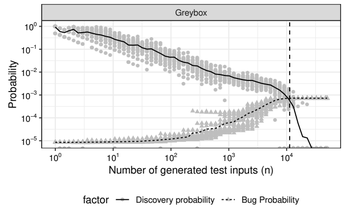


Figure 1: In greybox fuzzing, the probability p_{bug} to generate a bug-revealing input (dashed line) *increases* as n increases. The probability $\Delta(n)$ that the $(n+1)$-th input is coverage-increasing (solid line) provides an upper bound on the probability (residual risk) that it is the *first* bug-revealing input. The vertical line is when we expect the first bug-rev. input.

CCS CONCEPTS

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

KEYWORDS

software testing, statistics, estimation, assurance, correctness

ACM Reference Format:
Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468570

1 INTRODUCTION

On the one hand, we have software verification which allows to demonstrate the correctness of the program for *all inputs*. On the correctness of the program only for *some inputs*. While verification provides much stronger correctness guarantees, it is *greybox fuzzing*, a specific form of software testing, which has found widespread adoption in industry [24–26].

From a fuzzing campaign that has found no bugs, can we derive some statement about the correctness of the program? Fuzzing being a random process, it should be possible to derive statistical claims about the probability that the next generated input is the first bug-revealing input. We call this probability the *residual risk*. We know how to quantify residual risk for whitebox fuzzing (using model counting) [10] and blackbox fuzzing (using estimation) [1], but not for greybox fuzzing—which has emerged as the state-of-the-art in automated vulnerability discovery.

Greybox fuzzing is subject to *adaptive bias*, i.e., the probability to generate a bug-revealing input actually *increases* throughout the fuzzing campaign.[1] Figure 1 shows simulation results for greybox fuzzing. As more seeds become available, the bug probability p_{bug} increased (dashed line). In contrast, blackbox fuzzing is *not* subject to adaptive bias and the probability to generate a bug-revealing input remains constant throughout the campaign. If this was the case for greybox fuzzing, we could cast residual risk estimation as a *sunrise problem*[2] and employ the well-known Laplace estimator.

ICLR'25

How Much is Unseen Depends Chiefly on Information About the Seen

Seongmin Lee and Marcel Böhme
MPI for Security and Privacy, Germany
{seongmin.lee,marcel.boehme}@mpi-sp.org

Abstract

The *missing mass* refers to the proportion of data points in an *unknown* population of classifier inputs that belong to classes *not* present in the classifier's training data, which is assumed to be a random sample from that unknown population. We find that *in expectation* the missing mass is entirely determined by the number f_k of classes that *do* appear in the training data the same number of times *and an exponentially decaying error*. While this is the first precise characterization of the expected missing mass in terms of the sample, the induced estimator suffers from an impractically high variance. However, our theory suggests a large search space of nearly unbiased estimators that can be searched effectively and efficiently. Hence, we cast distribution-free estimation as an optimization problem to find a distribution-specific estimator with a minimized mean-squared error (MSE), given only the sample. In our experiments, our search algorithm discovers estimators that have a substantially smaller MSE than the state-of-the-art Good-Turing estimator. This holds for over 93% of runs when there are at least as many samples as classes. Our estimators' MSE is roughly 80% of the Good-Turing estimator's.

1 Introduction

How can we extrapolate from properties of the training data to properties of the unseen, underlying

0

TOSEM'18

STADS: Software Testing as Species Discovery

Spatial and Temporal Extrapolation from Tested Program Behaviors

MARCEL BÖHME*, National University of Singapore *and* Monash University, Australia

FSE'23

Statistical Reachability Analysis

Seongmin Lee
Max Planck Institute for Security and Privacy
Bochum, Germany
seongmin.lee@mpi-sp.org

Marcel Böhme
Max Planck Institute for Security and Privacy
Bochum, Germany
marcel.boehme@acm.org

ICSE'19

2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

Assurance in Software Testing: A Roadmap

Marcel Böhme
Monash University
marcel.boehme@acm.org

ICSE'24

Extrapolating Coverage Rate in Greybox Fuzzing

Danushka Liyanage*
Monash University
Australia

Seongmin Lee*
MPI-SP
Germany

Chakkrit Tantithamthavorn
Monash University
Australia

Marcel Böhme
MPI-SP
Germany

FSE'21

Estimating Residual Risk in Greybox Fuzzing

Marcel Böhme
Monash University, Australia
MPI-SP, Germany

Danushka Liyanage
Monash University
Australia

Valentin Wüstholz
ConsenSys
Germany

ABSTRACT

For any errorless fuzzing campaign, no matter how long, there is always some residual risk that a software error would be discovered if only the campaign was run for just a bit longer. Recently, greybox fuzzing tools have found widespread adoption. Yet, practitioners can only guess when the residual risk of a greybox fuzzing campaign falls below a specific, maximum allowable threshold.

In this paper, we explain why residual risk cannot be directly estimated for greybox campaigns, argue that the discovery probability (i.e., the probability that the next generated input increases code coverage) provides an excellent upper bound, and explore sound statistical methods to estimate the discovery probability in an ongoing greybox campaign. We find that estimators for blackbox fuzzing systematically and substantially *under*-estimate the true risk. An engineer—who stops the campaign when the estimators purport a risk below the maximum allowable risk—is vastly misled. She might need execute a campaign that is orders of magnitude longer to achieve the allowable risk. Hence, the *key challenge* we address in this paper is *adaptive bias*: The probability to discover a specific error actually increases over time. We provide the first probabilistic analysis of adaptive bias, and introduce two novel classes of estimators that tackle adaptive bias. With our estimators, the engineer can decide with confidence when to abort the campaign.

CCS CONCEPTS

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

KEYWORDS

software testing, statistics, estimation, assurance, correctness

ACM Reference Format:
Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468570

1 INTRODUCTION

On the one hand, we have software verification which allows to demonstrate the correctness of the program for *all inputs*. On the

Figure 1: In greybox fuzzing, the probability p_{bug} to generate a bug-revealing input (dashed line) *increases* as n increases. The probability $\Delta(n)$ that the $(n+1)$-th input is coverage-increasing (solid line) provides an upper bound on the probability (residual risk) that it is the *first* bug-revealing input. The vertical line is when we expect the first bug-rev. input.

correctness of the program only for *some inputs*. While verification provides much stronger correctness guarantees, it is *greybox fuzzing*, a specific form of software testing, which has found widespread adoption in industry [24–26].

From a fuzzing campaign that has found no bugs, can we derive some statement about the correctness of the program? Fuzzing being a random process, it should be possible to derive statistical claims about the probability that the next generated input is the first bug-revealing input. We call this probability the *residual risk*. We know how to quantify residual risk for whitebox fuzzing (using model counting) [10] and blackbox fuzzing (using estimation) [1], but not for greybox fuzzing—which has emerged as the state-of-the-art in automated vulnerability discovery.

Greybox fuzzing is subject to *adaptive bias*, i.e., the probability to generate a bug-revealing input actually *increases* throughout the fuzzing campaign.[1] Figure 1 shows simulation results for greybox fuzzing. As more seeds become available, the bug probability p_{bug} increased (dashed line). In contrast, blackbox fuzzing is *not* subject to adaptive bias and the probability to generate a bug-revealing input remains constant throughout the campaign. If this was the case for greybox fuzzing, we could cast residual risk estimation as a *sunrise problem*[2] and employ the well-known Laplace estimator.

ICLR'25

Why?

How Much is Unseen Depends Chiefly on Information About the Seen

Seongmin Lee and Marcel Böhme
MPI for Security and Privacy, Germany
{seongmin.lee,marcel.boehme}@mpi-sp.org

Abstract

The *missing mass* refers to the proportion of data points in an *unknown* population of classifier inputs that belong to classes *not* present in the classifier's training data, which is assumed to be a random sample from that unknown population. We find that *in expectation* the missing mass is entirely determined by the number f_k of classes that *do* appear in the training data the same number of times *and an exponentially decaying error*. While this is the first precise characterization of the expected missing mass in terms of the sample, the induced estimator suffers from an impractically high variance. However, our theory suggests a large search space of nearly unbiased estimators that can be searched effectively and efficiently. Hence, we cast distribution-free estimation as an optimization problem to find a distribution-specific estimator with a minimized mean-squared error (MSE), given only the sample. In our experiments, our search algorithm discovers estimators that have a substantially smaller MSE than the state-of-the-art Good-Turing estimator. This holds for over 93% of runs when there are at least as many samples as classes. Our estimators' MSE is roughly 80% of the Good-Turing estimator's.

1 Introduction

How can we extrapolate from properties of the training data to properties of the unseen, underlying

0 TOSEM'18

STADS: Software Testing as Species Discovery

Spatial and Temporal Extrapolation from Tested Program Behaviors

MARCEL BÖHME*, National University of Singapore *and* Monash University, Australia

2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

ICSE'

Assurance in Software Testing: A Roadmap

Marcel Böhme
Monash University
marcel.boehme@acm.org

FS

Estimating Residual Risk in Greybox Fuzzing

Marcel Böhme
Monash University, Australia
MPI-SP, Germany

Danushka Liyanage
Monash University
Australia

Valentin Wüstholz
ConsenSys
Germany

ABSTRACT

For any errorless fuzzing campaign, no matter how long, there is always some residual risk that a software error would be discovered if only the campaign was run for just a bit longer. Recently, greybox fuzzing tools have found widespread adoption. Yet, practitioners can only guess when the residual risk of a greybox fuzzing campaign falls below a specific, maximum allowable threshold.

In this paper, we explain why residual risk cannot be directly estimated for greybox campaigns, argue that the discovery probability (i.e., the probability that the next generated input increases code coverage) provides an excellent upper bound, and explore sound statistical methods to estimate the discovery probability in an ongoing greybox campaign. We find that estimators for blackbox fuzzing systematically and substantially *under*-estimate the true risk. An engineer—who stops the campaign when the estimators purport a risk below the maximum allowable risk—is vastly misled. She might need execute a campaign that is orders of magnitude longer to achieve the allowable risk. Hence, the *key challenge* we address in this paper is *adaptive bias*: The probability to discover a specific error actually increases over time. We provide the first probabilistic analysis of adaptive bias, and introduce two novel classes of estimators that tackle adaptive bias. With our estimators, the engineer can decide with confidence when to abort the campaign.

CCS CONCEPTS

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

KEYWORDS

software testing, statistics, estimation, assurance, correctness

ACM Reference Format:
Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468570

1 INTRODUCTION

On the one hand, we have software verification which allows to demonstrate the correctness of the program for *all inputs*. On the

Figure 1: In greybox fuzzing, the probability p_{bug} to generate a bug-revealing input (dashed line) *increases* as n increases. The probability $\Delta(n)$ that the $(n+1)$-th input is coverage-increasing (solid line) provides an upper bound on the probability (residual risk) that it is the *first* bug-revealing input. The vertical line is when we expect the first bug-rev. input.

correctness of the program only for *some inputs*. While verification provides much stronger correctness guarantees, it is *greybox fuzzing*, a specific form of software testing, which has found widespread adoption in industry [24–26].

From a fuzzing campaign that has found no bugs, can we derive some statement about the correctness of the program? Fuzzing being a random process, it should be possible to derive statistical claims about the probability that the next generated input is the first bug-revealing input. We call this probability the *residual risk*. We know how to quantify residual risk for whitebox fuzzing (using model counting) [10] and blackbox fuzzing (using estimation) [1], but not for greybox fuzzing—which has emerged as the state-of-the-art in automated vulnerability discovery.

Greybox fuzzing is subject to *adaptive bias*, i.e., the probability to generate a bug-revealing input actually *increases* throughout the fuzzing campaign.[1] Figure 1 shows simulation results for greybox fuzzing. As more seeds become available, the bug probability p_{bug} increased (dashed line). In contrast, blackbox fuzzing is *not* subject to adaptive bias and the probability to generate a bug-revealing input remains constant throughout the campaign. If this was the case for greybox fuzzing, we could cast residual risk estimation as a *sunrise problem*[2] and employ the well-known Laplace estimator.

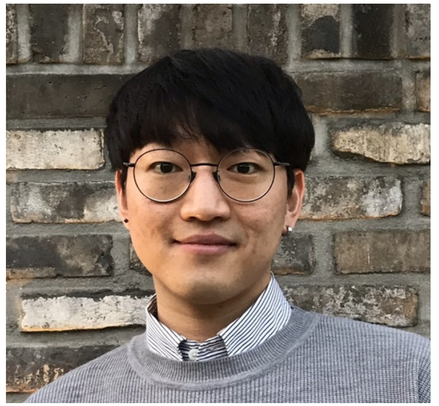
ICLR'25 Spotlight

How Much is Unseen Depends Chiefly on Information About the Seen

Seongmin Lee and Marcel Böhme
MPI for Security and Privacy, Germany
{seongmin.lee,marcel.boehme}@mpi-sp.org

Abstract

The *missing mass* refers to the proportion of data point
tion of classifier inputs that belong to classes *not* present
data, which is assumed to be a random sample from that
find that *in expectation* the missing mass is entirely de
f_k of classes that *do* appear in the training data the sa
an exponentially decaying error. While this is the first
of the expected missing mass in terms of the sample, th
fers from an impractically high variance. However, ou
search space of nearly unbiased estimators that can be
efficiently. Hence, we cast distribution-free estimation
lem to find a distribution-specific estimator with a minin
(MSE), given only the sample. In our experiments, our se
estimators that have a substantially smaller MSE than t
Turing estimator. This holds for over 93% of runs when
samples as classes. Our estimators' MSE is roughly 8
estimator's.



Seongmin Lee
MPI-SP

1 Introduction

How can we extrapolate from properties of the training data to pr
distribution of the data? This is a fundamental question in machi
Orlitsky & Suresh, 2015; Painsky, 2022; Acharya et al., 2013; Hac
a data point belongs to a class that does *not* exist in the training
probability mass since *empirically* the entire probability mass is di
in the training data. For instance, the missing mass measures ho
is of the unknown distribution. If the missing mass is high, the t
and a trained classifier is unlikely to predict the correct class. If w
missing mass also measures *saturation*. We may decide that the
and saturation has been reached when the missing mass is below

0 TOSEM'18

STADS: Software Testing as Species Discovery

Spatial and Temporal Extrapolation from Tested Program Behaviors

MARCEL BÖHME*, National University of Singapore *and* Monash University, Australia

2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)

ICSE'

Assurance in Software Testing: A Roadmap

Marcel Böhm
Monash Universit
marcel.boehme@acm

Estimating Re... R

Marcel Böhme
Monash University, Australia
MPI-SP, Germany

Danushk
Monash
Au

ABSTRACT

For any errorless fuzzing campaign, no matter how long, there is always some residual risk that a software error would be discovered if only the campaign was run for just a bit longer. Recently, greybox fuzzing tools have found widespread adoption. Yet, practitioners can only guess when the residual risk of a greybox fuzzing campaign falls below a specific, maximum allowable threshold.

In this paper, we explain why residual risk cannot be directly estimated for greybox campaigns, argue that the discovery probability (i.e., the probability that the next generated input increases code coverage) provides an excellent upper bound, and explore sound statistical methods to estimate the discovery probability in an ongoing greybox campaign. We find that estimators for blackbox fuzzing systematically and substantially *under*-estimate the true risk. An engineer—who stops the campaign when the estimators purport a risk below the maximum allowable risk—is vastly misled. She might need execute a campaign that is orders of magnitude longer to achieve the allowable risk. Hence, the *key challenge* we address in this paper is *adaptive bias*: The probability to discover a specific error actually increases over time. We provide the first probabilistic analysis of adaptive bias, and introduce two novel classes of estimators that tackle adaptive bias. With our estimators, the engineer can decide with confidence when to abort the campaign.

CCS CONCEPTS

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

KEYWORDS

software testing, statistics, estimation, assurance, correctness

ACM Reference Format:
Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468570

1 INTRODUCTION

On the one hand, we have software verification which allows to demonstrate the correctness of the program for *all inputs*. On the

🏆 **ICLR'25 Spotlight**

How Much is Unseen Depends Chiefly on Information About the Seen

Seongmin Lee and Marcel Böhme
MPI for Security and Privacy, Germany
{seongmin.lee,marcel.boehme}@mpi-sp.org

SBFT 2025

Mon 28 Apr 2025 14:30 - 15:30 at 104 - Paper Presentations 2 and Tutorial 1 Chair(s): Alessio Gambi

☆ Tutorial by Seongmin Lee

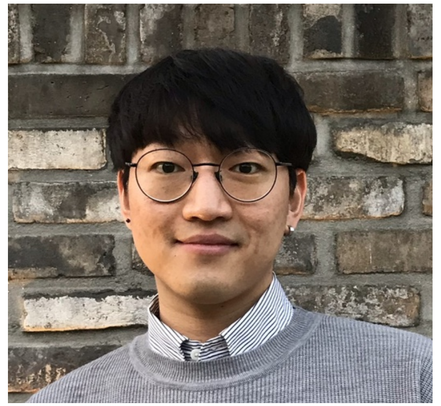
The Magic of Statistics for Software Testing: How to Foresee the Unseen

Ensuring software correctness is essential as software increasingly governs critical aspects of modern life. Formal methods for program verification, while powerful, often struggle with scalability when faced with the complexity of modern systems. Meanwhile, software testing—finding defects by executing the program—is practical but inherently incomplete, as it inevitably misses certain behaviors, i.e., the "unseens," leaving critical gaps in verification.

In this tutorial, I illuminate the transformative potential of statistical methods in addressing these challenges, with a particular focus on residual risk analysis. Residual risk analysis quantifies the likelihood of undiscovered bugs remaining in the software after testing by estimating the probability of finding a new, previously unseen bug in the next test input.

We will begin by demonstrating how statistical estimators can assess residual risk using records from software testing—such as code coverage data—through a hands-on example. The tutorial then explores several advanced extensions to adapt residual risk analysis for more realistic testing scenarios. By the end of this session, participants will gain a deeper understanding of how statistical thinking can provide actionable insights into the unseen behaviors of software systems, ultimately making testing more accountable, transparent, and efficient.

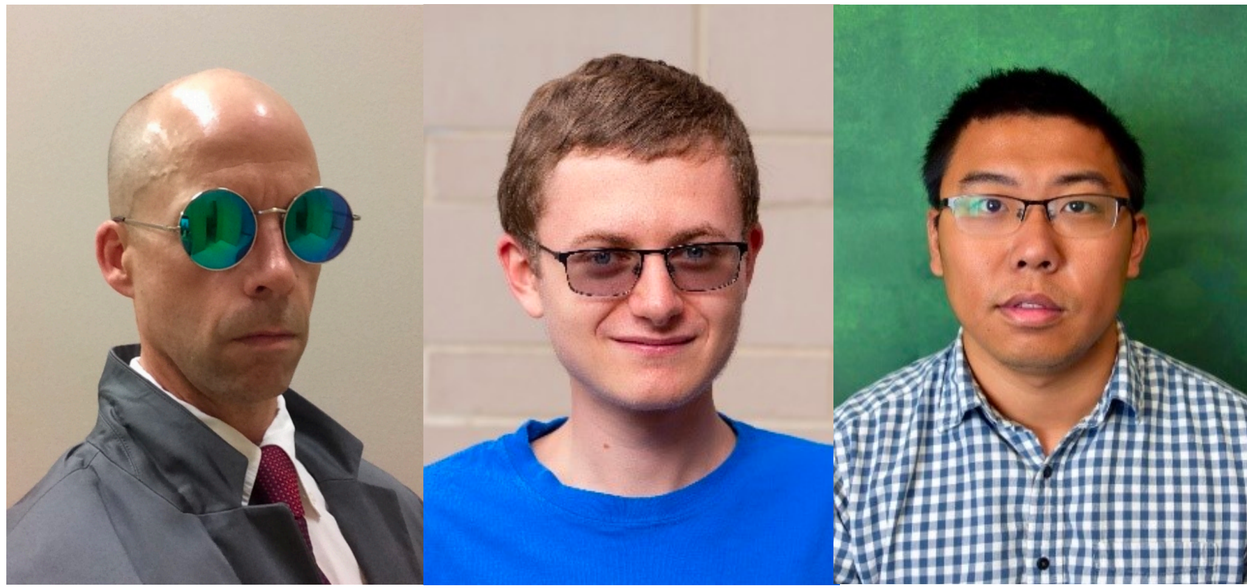
Seongmin Lee
Max Planck Institute for Security and Privacy (MPI-SP)
Germany



Seongmin Lee
MPI-SP

Correlation: Very strong

- Problem:
  - Fuzzing folks are still not convinced that coverage is a good measure.



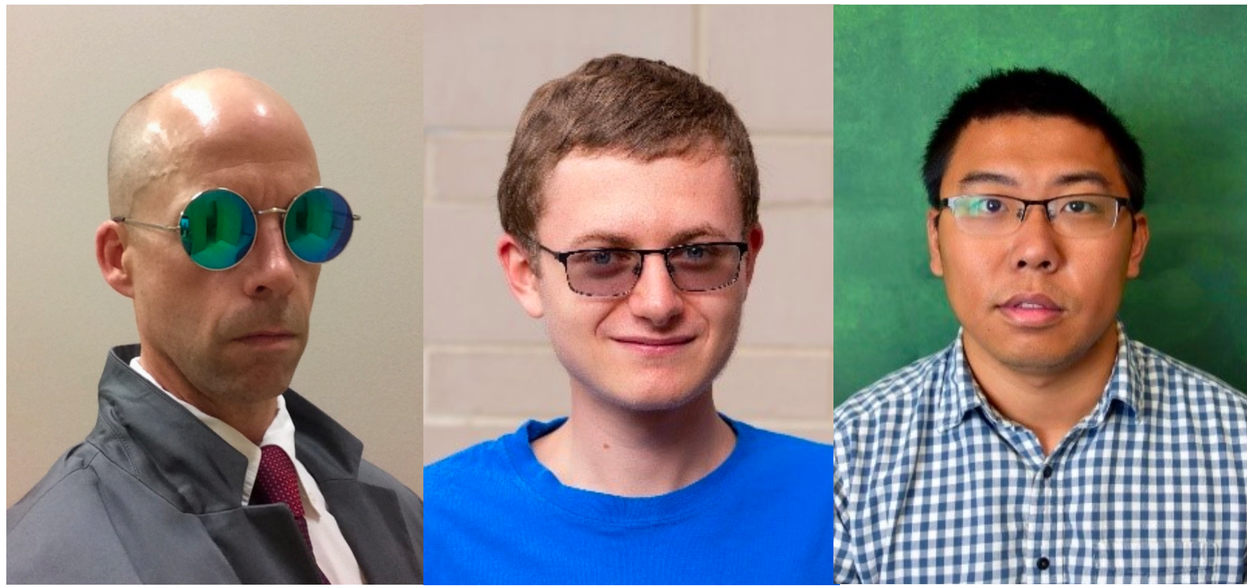
“It does not make sense 🤔”

paraphrasing Klees et al., CCS’18

Correlation: Very strong

- Problem:
  - Fuzzing folks are still not convinced that coverage is a good measure.

"We cannot compare two or more fuzzers in terms of coverage in order to establish one as the best in terms of bug finding.



"It does not make sense 🤔"

paraphrasing Klees et al., CCS'18

Correlation: Very strong

- Problem:
  - Fuzzing folks are still not convinced that coverage is a good measure.

Session 10D: VulnDet 2 + Side Channels 2 CCS'18, October 15-19, 2018, Toronto, ON, Canada

Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Maryland

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzer algorithm (call it *A*), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling *baseline* fuzzer *B* to compare against;
- a sample of target programs—the *benchmark suite*;
- a *performance metric* to measure when *A* and *B* are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of *configuration parameters*, e.g., the *seed file* (or files) to start fuzzing with, and the *timeout* (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random

One solution is to instead (or also) measure the improvement in code coverage made by fuzzer *A* over baseline *B*. Greybox fuzzers already aim to optimize coverage as part of the `isInteresting` function, so surely showing an improved code coverage would indicate an improvement in fuzzing. This makes sense. To find a crash at a particular point in the program, that point in the program would need to execute. Prior studies of test suite effectiveness also suggest that higher coverage correlates with bug finding effectiveness [19, 30]. Nearly half of the papers we considered measured code coverage; FairFuzz *only* evaluated performance using code (branch) coverage [32].

However, there is no *fundamental* reason that maximizing code coverage is directly connected to finding bugs. While the general efficacy of coverage-guided fuzzers over black box ones implies that there's a strong correlation, particular algorithms may eschew higher coverage to focus on other signs that a bug may be present. For example, AFLGo [5] does not aim to increase coverage globally, but rather aims to focus on particular, possibly error-prone points in the program. Even if we assume that coverage and bug finding are correlated, that correlation may be weak [28]. As such, a substantial improvement in coverage may yield merely a negligible improvement in bug finding effectiveness.

Correlation: Very strong

- Problem:
  - Fuzzing folks are still not convinced that coverage is a good measure.

Agreement: That’s why.

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Address for correspondence: Dr. Priya Ranganathan, Department of Anaesthesiology, Tata Memorial Centre, Ernest Borges Road, Parel, Mumbai - 400 012, Maharashtra, India.
E-mail: drpriyaranganathan@gmail.com

INTRODUCTION

Often, one is interested in knowing whether measurements made by two (sometimes more than two) different observers or by two different techniques produce similar results. This is referred to as agreement or concordance or reproducibility between measurements. Such analysis looks at pairs of measurements, either both categorical or both numeric, with each pair having been made on one individual (or a pathology slide, or an X-ray).

Superficially, these data may appear to be amenable to analysis using methods used for 2 × 2 tables (if the variable is categorical) or correlation (if numeric), which we have discussed previously in this series.[1,2] However, a closer look would show that this is not true. In those methods, two measurements relate to the same variable (e.g., chest radiographs rated by two radiologists or hemoglobin measured by two methods).

WHAT IS AGREEMENT?

Let us consider the case of two examiners A and B evaluating answer sheets of 20 students in a class and marking each of them as "pass" or "fail," with each examiner passing half the students. Table 1 shows three different situations that may happen. In situation 1 in this table, eight students receive a "pass" grade from both the examiners, eight receive a "fail" grade from both the examiners, and four receive pass grade from one examiner but "fail" grade from the other (two passed by A and the other two by B). Thus, the two examiners' results

POINTS TO REMEMBER

Correlation versus agreement

As alluded to above, correlation is not synonymous with agreement. Correlation refers to the presence of a relationship between two different variables, whereas agreement looks at the concordance between two measurements of one variable. Two sets of observations, which are highly correlated, may have poor agreement; however, if the two sets of values agree, they will surely be highly correlated. For instance, in the hemoglobin example, even though the agreement is poor, the correlation coefficient between values from the two methods is high [Figure 2]; ($r = 0.98$). The other way to look at it is that, though the individual dots are not fairly close to the dotted line (least square line;[2] indicating good correlation), these are quite far from the solid black line, which represents the line of perfect agreement (Figure 2: the solid black line). In case of good agreement, the dots would be expected to fall on or near this (the solid black) line.

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Address for correspondence: Dr. Priya Ranganathan, Department of Anaesthesiology, Tata Memorial Centre, Ernest Borges Road, Parel, Mumbai - 400 012, Maharashtra, India.
E-mail: drpriyaranganathan@gmail.com

INTRODUCTION

Often, one is interested in knowing whether measurements made by two (sometimes more than two) different observers or by two different techniques produce similar results. This is referred to as agreement or concordance or reproducibility between measurements. Such analysis looks at pairs of measurements, either both categorical or both numeric, with each pair having been made on one individual (or a pathology slide, or an X-ray).

Superficially, these data may appear to be amenable to analysis using methods used for 2 × 2 tables (if the variable is categorical) or correlation (if numeric), which we have discussed previously in this series.[1,2] However, a closer look would show that this is not true. In those methods, two measurements relate to the same variable (e.g., chest radiographs rated by two radiologists or hemoglobin measured by two methods).

WHAT IS AGREEMENT?

Let us consider the case of two examiners A and B evaluating answer sheets of 20 students in a class and marking each of them as "pass" or "fail," with each examiner passing half the students. Table 1 shows three different situations that may happen. In situation 1 in this table, eight students receive a "pass" grade from both the examiners, eight receive a "fail" grade from both the examiners, and four receive pass grade from one examiner but "fail" grade from the other (two passed by A and the other two by B). Thus, the two examiners' results

- Suppose, we have
  Two instruments to measure acidity.

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Address for correspondence: Dr. Priya Ranganathan, Department of Anaesthesiology, Tata Memorial Centre, Ernest Borges Road, Parel, Mumbai - 400 012, Maharashtra, India.
E-mail: drpriyaranganathan@gmail.com

INTRODUCTION

Often, one is interested in knowing whether measurements made by two (sometimes more than two) different observers or by two different techniques produce similar results. This is referred to as agreement or concordance or reproducibility between measurements. Such analysis looks at pairs of measurements, either both categorical or both numeric, with each pair having been made on one individual (or a pathology slide, or an X-ray).

Superficially, these data may appear to be amenable to analysis using methods used for 2 × 2 tables (if the variable is categorical) or correlation (if numeric), which we have discussed previously in this series.[1,2] However, a closer look would show that this is not true. In those methods, two measurements relate to the same variable (e.g., chest radiographs rated by two radiologists or hemoglobin measured by two methods).

WHAT IS AGREEMENT?

Let us consider the case of two examiners A and B evaluating answer sheets of 20 students in a class and marking each of them as "pass" or "fail," with each examiner passing half the students. Table 1 shows three different situations that may happen. In situation 1 in this table, eight students receive a "pass" grade from both the examiners, eight receive a "fail" grade from both the examiners, and four receive pass grade from one examiner but "fail" grade from the other (two passed by A and the other two by B). Thus, the two examiners' results

- Two instruments to measure acidity.
- Strong correlation:
  - More acidity = both indicate higher PH values.

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Address for correspondence: Dr. Priya Ranganathan, Department of Anaesthesiology, Tata Memorial Centre, Ernest Borges Road, Parel, Mumbai - 400 012, Maharashtra, India.
E-mail: drpriyaranganathan@gmail.com

INTRODUCTION

Often, one is interested in knowing whether measurements made by two (sometimes more than two) different observers or by two different techniques produce similar results. This is referred to as agreement or concordance or reproducibility between measurements. Such analysis looks at pairs of measurements, either both categorical or both numeric, with each pair having been made on one individual (or a pathology slide, or an X-ray).

Superficially, these data may appear to be amenable to analysis using methods used for 2 × 2 tables (if the variable is categorical) or correlation (if numeric), which we have discussed previously in this series.[1,2] However, a closer look would show that this is not true. In those methods, two measurements relate to the same variable (e.g., chest radiographs rated by two radiologists or hemoglobin measured by two methods).

WHAT IS AGREEMENT?

Let us consider the case of two examiners A and B evaluating answer sheets of 20 students in a class and marking each of them as "pass" or "fail," with each examiner passing half the students. Table 1 shows three different situations that may happen. In situation 1 in this table, eight students receive a "pass" grade from both the examiners, eight receive a "fail" grade from both the examiners, and four receive pass grade from one examiner but "fail" grade from the other (two passed by A and the other two by B). Thus, the two examiners' results

- Two instruments to measure acidity.
- Strong correlation:
  - More acidity = both indicate higher PH values.
- Weak agreement:
  - Both instruments might rank 2+ tubes differently.

Agreement: That's why.

Statistics

Common pitfalls in statistical analysis: Measures of agreement

Priya Ranganathan, C. S. Pramesh[1], Rakesh Aggarwal[2]

Departments of Anaesthesiology and [1]Surgical Oncology, Tata Memorial Centre, Mumbai, Maharashtra, [2]Department of Gastroenterology, Sanjay Gandhi Postgraduate Institute of Medical Sciences, Lucknow, Uttar Pradesh, India

Abstract Agreement between measurements refers to the degree of concordance between two (or more) sets of measurements. Statistical methods to test agreement are used to assess inter-rater variability or to decide whether one technique for measuring a variable can substitute another. In this article, we look at statistical measures of agreement for different types of data and discuss the differences between these and those for assessing correlation.

Keywords: Agreement, biostatistics, concordance

Address for correspondence: Dr. Priya Ranganathan, Department of Anaesthesiology, Tata Memorial Centre, Ernest Borges Road, Parel, Mumbai - 400 012, Maharashtra, India.
E-mail: drpriyaranganathan@gmail.com

INTRODUCTION

Often, one is interested in knowing whether measurements made by two (sometimes more than two) different observers or by two different techniques produce similar results. This is referred to as agreement or concordance or reproducibility between measurements. Such analysis looks at pairs of measurements, either both categorical or both numeric, with each pair having been made on one individual (or a pathology slide, or an X-ray).

Superficially, these data may appear to be amenable to analysis using methods used for 2 × 2 tables (if the variable is categorical) or correlation (if numeric), which we have discussed previously in this series.[1,2] However, a closer look would show that this is not true. In those methods, two measurements relate to the same variable (e.g., chest radiographs rated by two radiologists or hemoglobin measured by two methods).

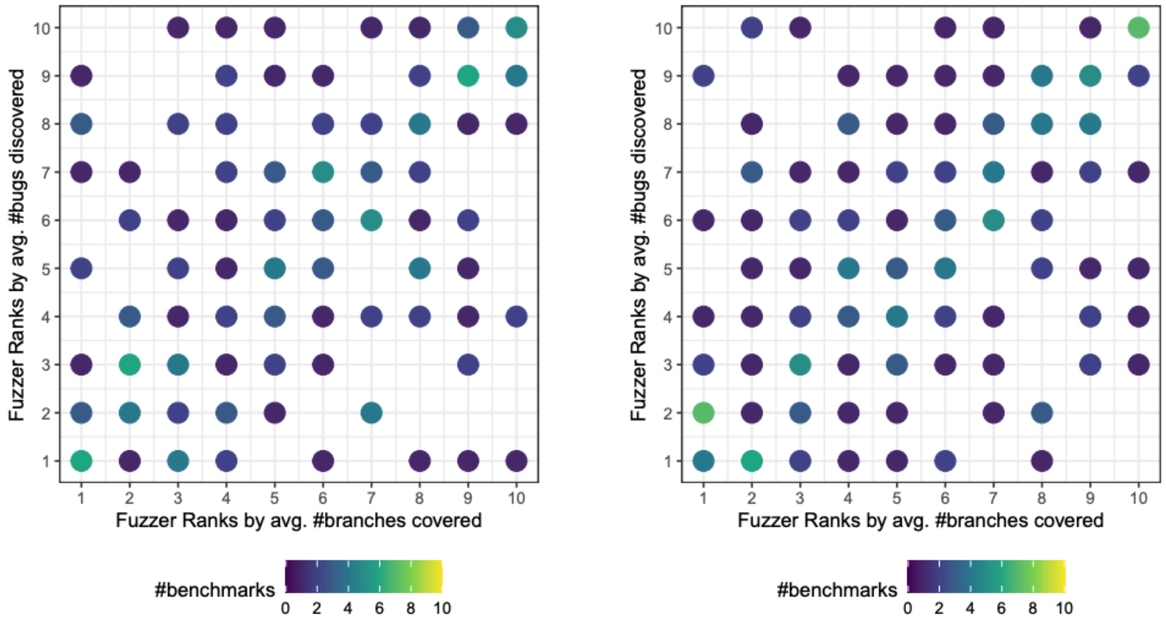
WHAT IS AGREEMENT?

Let us consider the case of two examiners A and B evaluating answer sheets of 20 students in a class and marking each of them as "pass" or "fail," with each examiner passing half the students. Table 1 shows three different situations that may happen. In situation 1 in this table, eight students receive a "pass" grade from both the examiners, eight receive a "fail" grade from both the examiners, and four receive pass grade from one examiner but "fail" grade from the other (two passed by A and the other two by B). Thus, the two examiners' results

Moderate agreement means we cannot reliably substitute one instrument for the other.

- Weak agreement:
  - Both instruments might rank 2+ tubes differently.

Agreement: Coverage vs Bug Finding



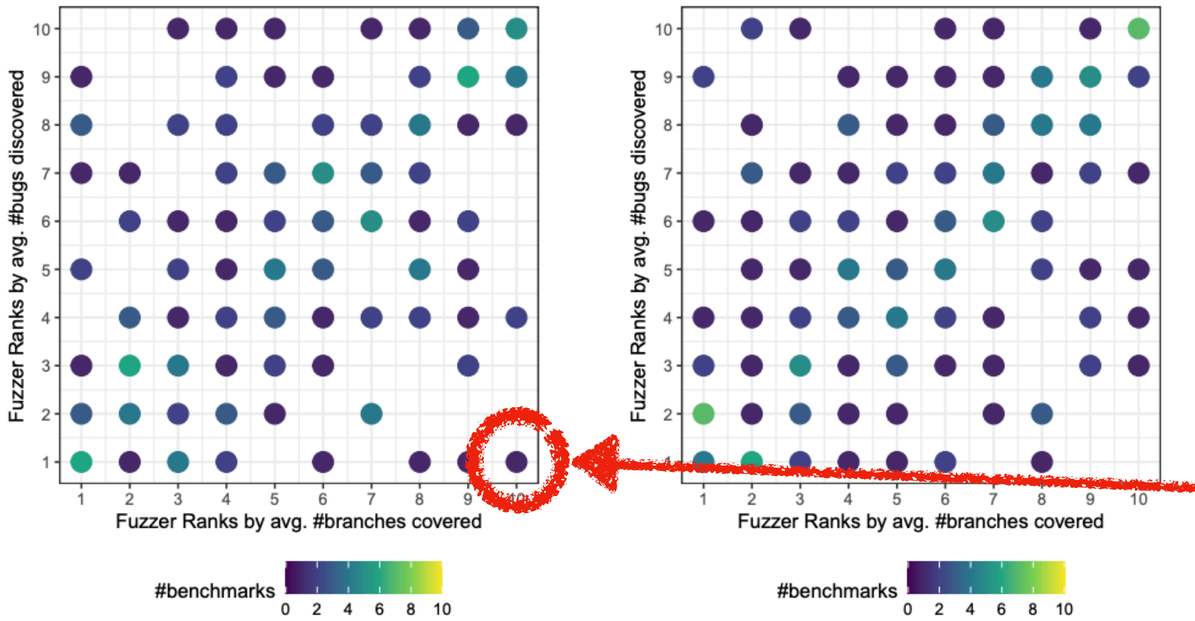
(a) 1 hour fuzzing campaigns ($\rho = 0.38$). **(b)** 1 day fuzzing campaigns ($\rho = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Ranking 10 fuzzers
in terms of code coverage and
in terms of #bugs found.

Moderate agreement means
we cannot reliably substitute
one instrument for the other.

Agreement: Coverage vs Bug Finding



(a) 1 hour fuzzing campaigns ($\rho = 0.38$). **(b)** 1 day fuzzing campaigns ($\rho = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Ranking 10 fuzzers
in terms of code coverage and
in terms of #bugs found.

The worst fuzzer in terms coverage is the best fuzzer in terms of bug finding.

Moderate agreement means we cannot reliably substitute one instrument for the other.

Agreement: Coverage vs Bug Finding

- 10 fuzzers x 24 random open source projects x 23h x 20 trials
- We observe a moderate agreement on superiority or ranking.
- Only if we require differences in coverage *and* bug finding to be highly statistically significant, we observe a strong agreement.

You can substitute
coverage for bug finding
only with moderate reliability.

Search-Based and Fuzz Testing (SBFT)'23

Fuzzing Competition

Abhishek Arya

GOOGLE, USA

Dongge Liu

GOOGLE, USA

Jonathan Metzman

GOOGLE, USA

Marcel Böhme

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY, GERMANY

Oliver Chang

GOOGLE, USA

Goals

- Promote innovative fuzzers in software vulnerability discovery
- Encourage developers and researchers to present and discuss their work
- Contribute a free and easy-to-use infrastructure for the community

53 Benchmarks

	Coverage-based	Bug-based
Public	24	5
Hidden	14	10

Coverage-based Ranking

fuzzer	
hastefuzz	97.51
aflplusplusplus	94.19
aflplusplus	93.46
aflrustrust	91.82
libafl_libfuzzer	89.88
honggfuzz	88.08
pastis	86.70
libfuzzer	83.43
symsan	79.09
afl	75.60
aflsmart_plusplus	66.60
learnperffuzz	42.95

Bug-based Ranking

fuzzer		
pastis	53.33	0.0
aflrustrust	53.33	960.0
aflsmart_plusplus	50.00	1440.0
afl	46.67	4140.0
honggfuzz	46.67	5310.0
libafl_libfuzzer	46.67	5490.0
aflplusplusplus	40.00	7680.0
hastefuzz	40.00	8880.0
libfuzzer	40.00	9030.0
aflplusplus	40.00	9600.0
symsan	20.00	24720.0
learnperffuzz	6.67	32100.0

Don’t measure coverage. Measure bugs?

- Posthoc bug-based evaluation
  - Choose a random, representative sample of programs and fuzz them.
    - (Un)fortunately, bugs are very sparse. No statistical power.
    - Maximize bug probability to for economical reasons.
  - Identify and deduplicate bugs *after* the fuzzing campaign. Minimizes bias.
  - Problem: Less economical (we did not find bugs in 7/24 [30%] programs).

Don’t measure coverage. Measure bugs?

- Posthoc bug-based evaluation
  - Choose a random, representative sample of programs and fuzz them.
    - (Un)fortunately, bugs are very sparse. No statistical power.
    - Maximize bug probability to for economical reasons.
  - Identify and deduplicate bugs *after* the fuzzing campaign. Minimizes bias.
  - Problem: Less economical (we did not find bugs in 7/24 [30%] programs).

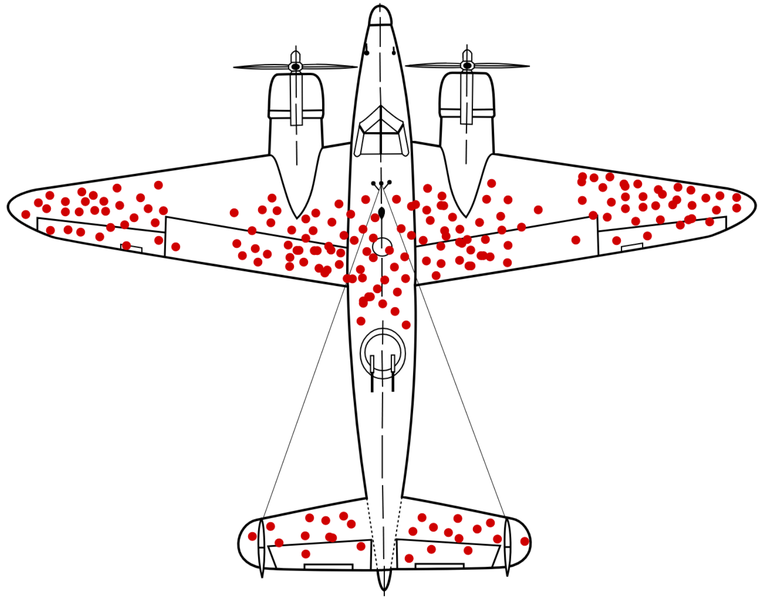
- Mutation-based evaluation
  - Inject synthetic bugs into a random, representative sample of programs
  - More economical. We know many bugs can be found.
  - Problem: Are synthetic bugs representative of real bugs?

Don’t measure coverage. Measure bugs?

- Ground-truth-based evaluation
  - Curate real bugs in a random, representative sample of programs.
  - Economical, realistic bugs, objective ground truth.

Don’t measure coverage. Measure bugs?

- Ground-truth-based evaluation
  - Curate real bugs in a random, representative sample of programs.
  - Economical, realistic bugs, objective ground truth.
  - Problem:
    1. Survivorship bias
       - Fuzzers that are better at finding previously undiscovered bugs appear worse.
       - Fuzzers that contributed to the original discovery appear better.



Don’t measure coverage. Measure bugs?

- Ground-truth-based evaluation
  - Curate real bugs in a random, representative sample of programs.
  - Economical, realistic bugs, objective ground truth.
  - Problem:
    1. Survivorship bias
    2. Confirmation bias
       - Given a ground truth benchmark, researchers might be enticed to iteratively and unknowingly tune their fuzzer to the benchmark.

17 October 2019

When Results Are All That Matters: Consequences

by Andreas Zeller and Sascha Just; with Kai Greshake

6. Researchers must resist the temptation of optimizing their tools towards a specific benchmark.

While developing an approach, it is only natural to try it out on some examples to assess its performance, such that results may guide further refinement. The risk of such guidance, however, is that development may result in overspecialization – i.e., an approach that works well on a benchmark, but not on other programs. As a result, one will get a paper without impact and a tool that nobody uses.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

→ Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

→ Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

→ Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Measures are specific, our claims general.

ICSE'22

On the Reliability of Coverage-Based Fuzzer Benchmarking

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

László Szekeres
Google, USA

Jonathan Metzman
Google, USA

ABSTRACT

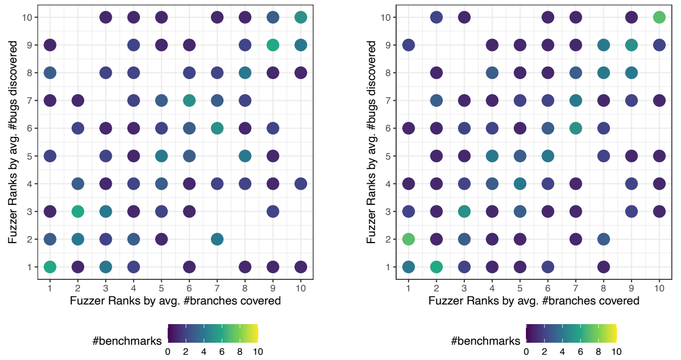
Given a program where none of our fuzzers finds any bugs, how do we know which fuzzer is better? In practice, we often look to code coverage as a proxy measure of fuzzer effectiveness and consider the fuzzer which achieves more coverage as the better one.

Indeed, evaluating 10 fuzzers for 23 hours on 24 programs, we find that a fuzzer that covers more code also finds more bugs. There is a *very strong correlation* between the coverage achieved and the number of bugs found by a fuzzer. Hence, it might seem reasonable to compare fuzzers in terms of coverage achieved, and from that derive empirical claims about a fuzzer's superiority at finding bugs.

Curiously enough, however, we find *no strong agreement* on which fuzzer is superior if we compared multiple fuzzers in terms of coverage achieved instead of the number of bugs found. The fuzzer best at achieving coverage, may not be best at finding bugs.

ACM Reference Format:
Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510230

1 INTRODUCTION



(a) 1 hour fuzzing campaigns ($\rho = 0.38$). **(b)** 1 day fuzzing campaigns ($\rho = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Hence, it might seem reasonable to conjecture that the fuzzer which is better in terms of code coverage is also better in terms of bug finding—but is this really true? In Figure 1, we show the ranking of these fuzzers across all programs in terms of the average

Jonathan Metzman
Google

László Szekeres
Google

Marcel E

Benchmarks are specific, our claims general.

TOSEM'25

Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome*. This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization*, can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

Dylan Wolff
NUS

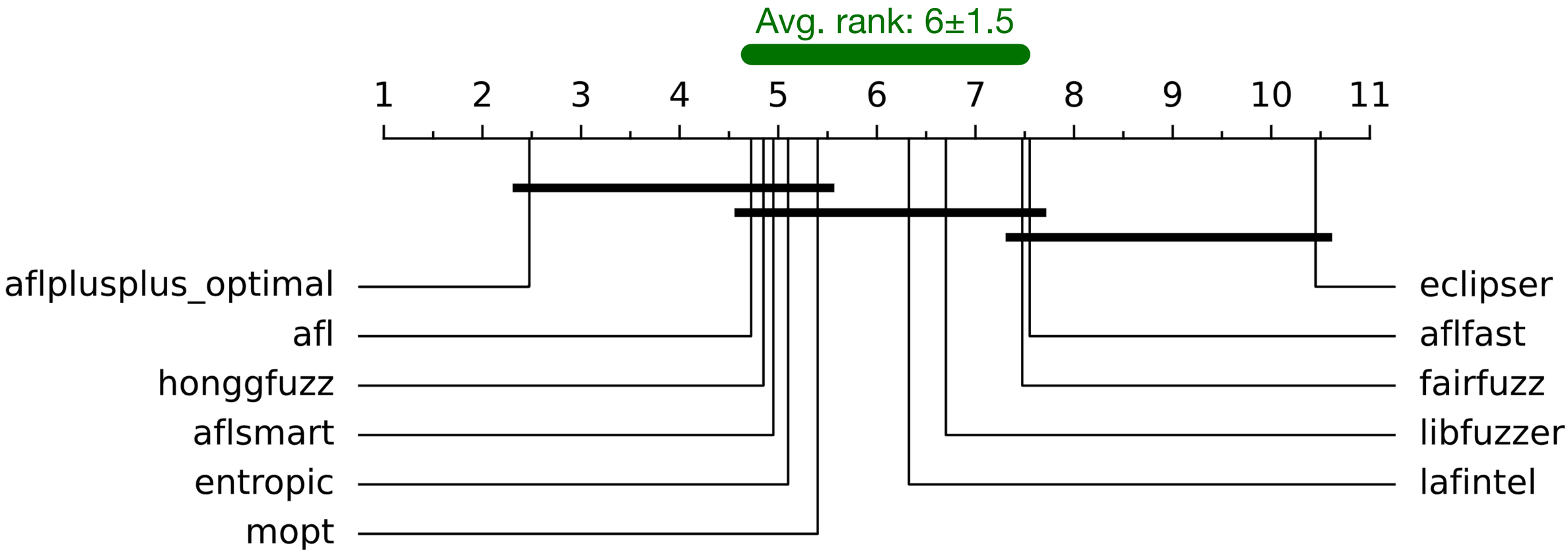
Abhik Roychoudhury
NUS

Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.

Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.



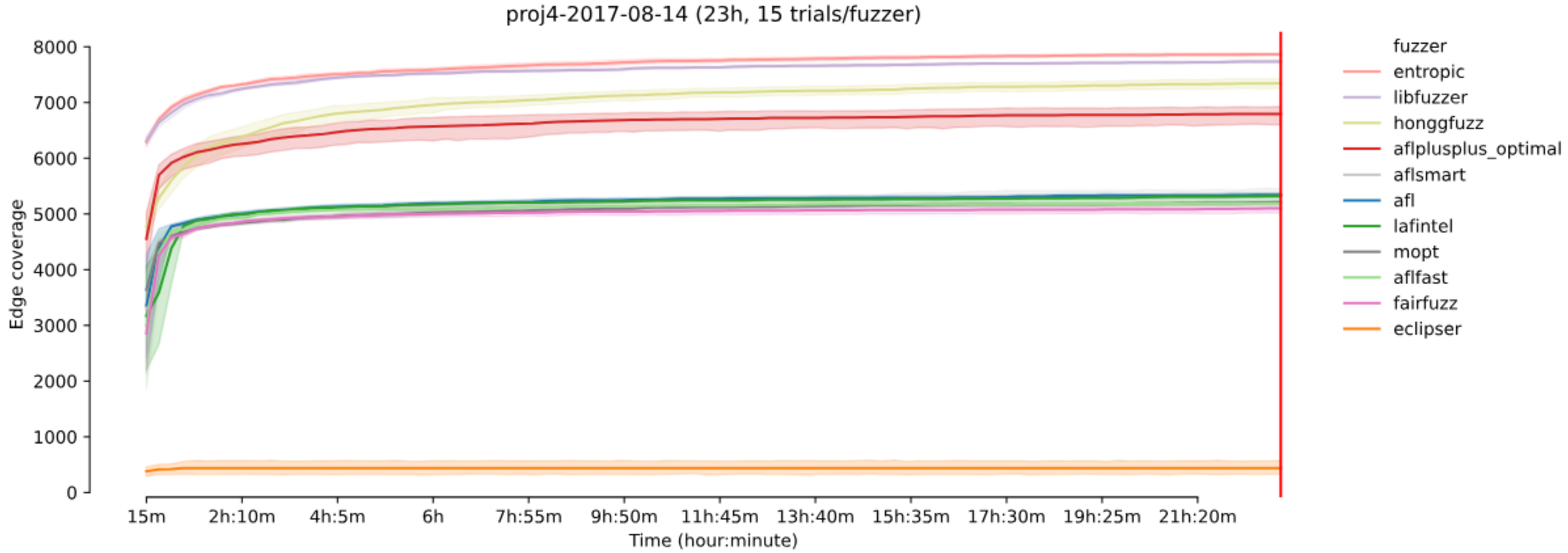
Source: 2020 Sample Fuzzbench Report (https://www.fuzzbench.com/reports/sample/index.html)

Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.
  - For each specific program, there are clear winners.

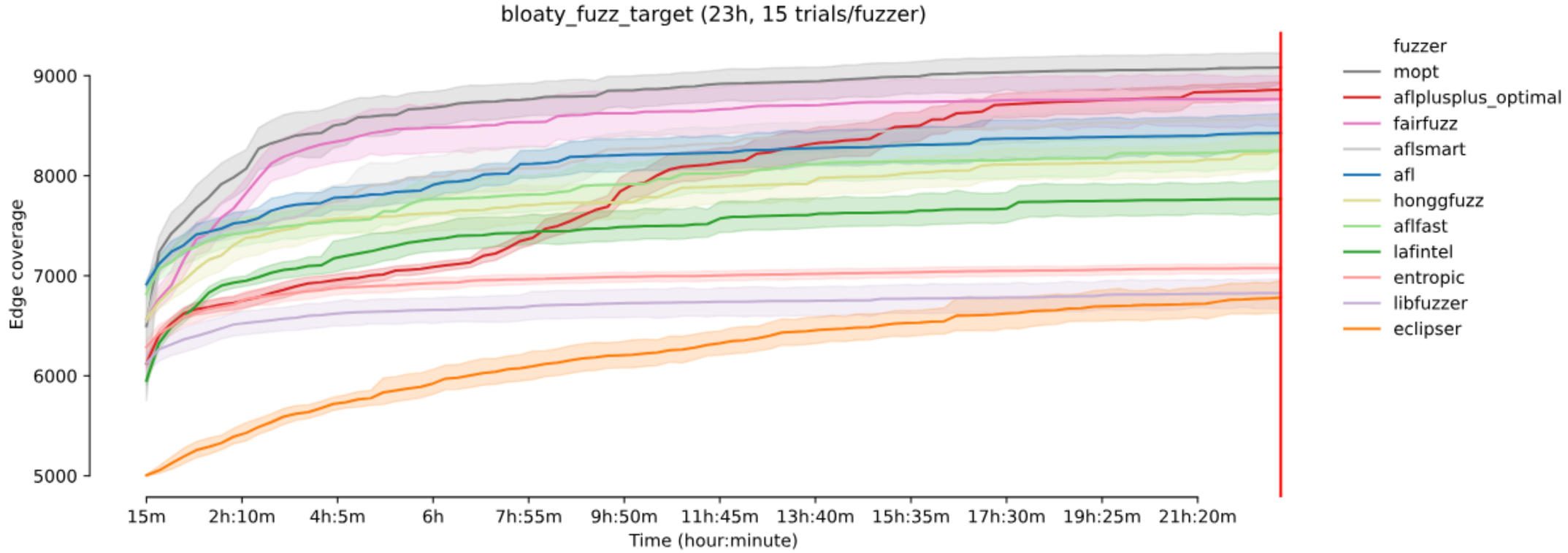
Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.
  - For each specific program, there are clear winners.



Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.
  - For each specific program, there are clear winners.



Benchmarks are specific, our claims general.

- Observation:
  - On the average, most fuzzers perform similarly.
  - For each specific program, there are clear winners.
- Atomistic benchmarking doesn't show that, e.g., the ranking of AFL++ improves on larger programs.

By avg. score

fuzzer	average normalized score
aflplusplus_optimal	98.61
honggfuzz	95.37
entropic	93.75
lafintel	91.53
libfuzzer	91.47
aflsmart	90.35
afl	89.89
mopt	89.60
aflfast	87.67
fairfuzz	84.74
eclipser	76.68

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome
  is a function of the specific benchmark properties.

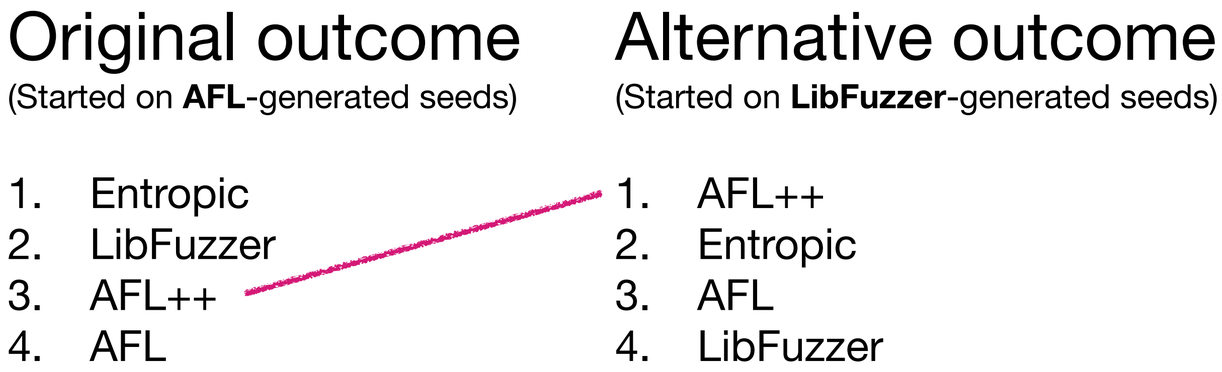
Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome
  is a function of the specific benchmark properties.

- We propose a counterfactual analysis
  - to report the conditions under which the benchmark outcome would change.
  - to quantify the impact of a change in a benchmark property on the outcome.

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis
  - to report the conditions under which the benchmark outcome would change.
  - to quantify the impact of a change in a benchmark property on the outcome.
- **Experiment**:
  - Manipulate one property.
  - Report difference in ranking.



Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.

- We propose a counterfactual analysis
  - to report the conditions under which the benchmark outcome would change.
  - to quantify the impact of a change in a benchmark property on the outcome.

- **Randomization**:
  - Manipulate many properties.
  - Report multiple linear regression.

$$R = \alpha + \left[\sum_{p_i \in P} \beta_i X_i\right] + \left[\sum_{f \in F} \gamma_f Y_f\right] + \left[\sum_{p_i \in P} \sum_{f \in F} \omega_{i,f} X_i Y_f\right]$$

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis

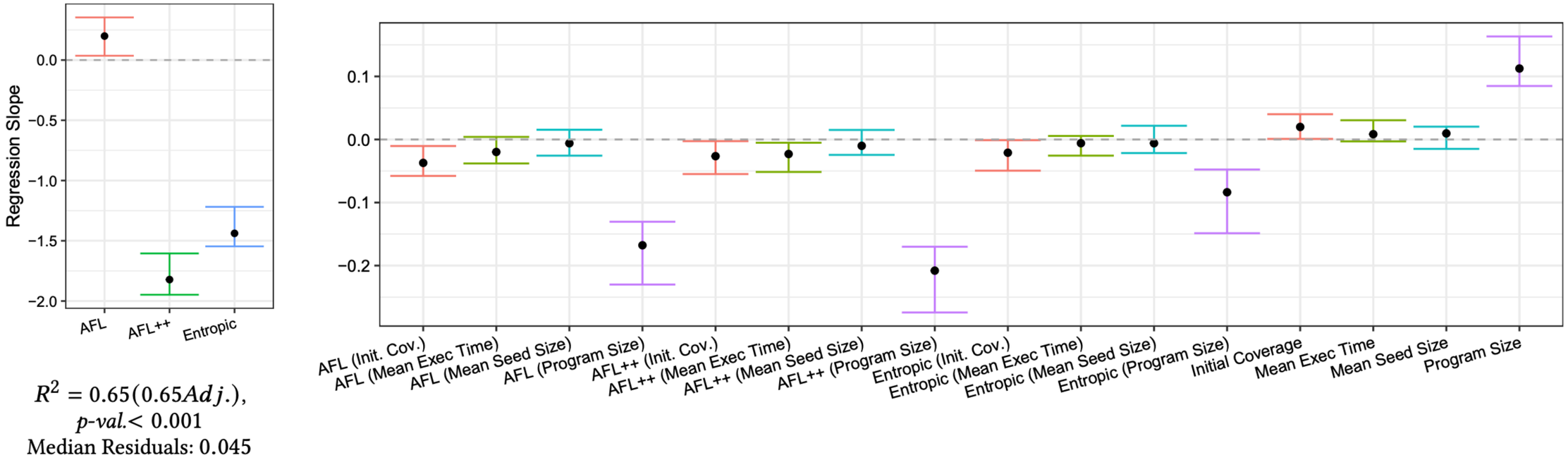
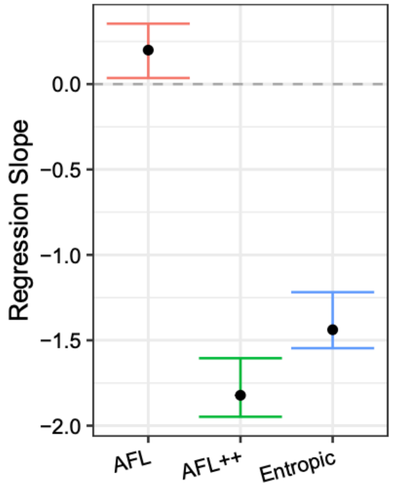


Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking ~ Fuzzer × Properties) [Eqn. 1].

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis



$R^2 = 0.65 (0.65 Adj.)$,
$p\text{-}val. < 0.001$
Median Residuals: 0.045

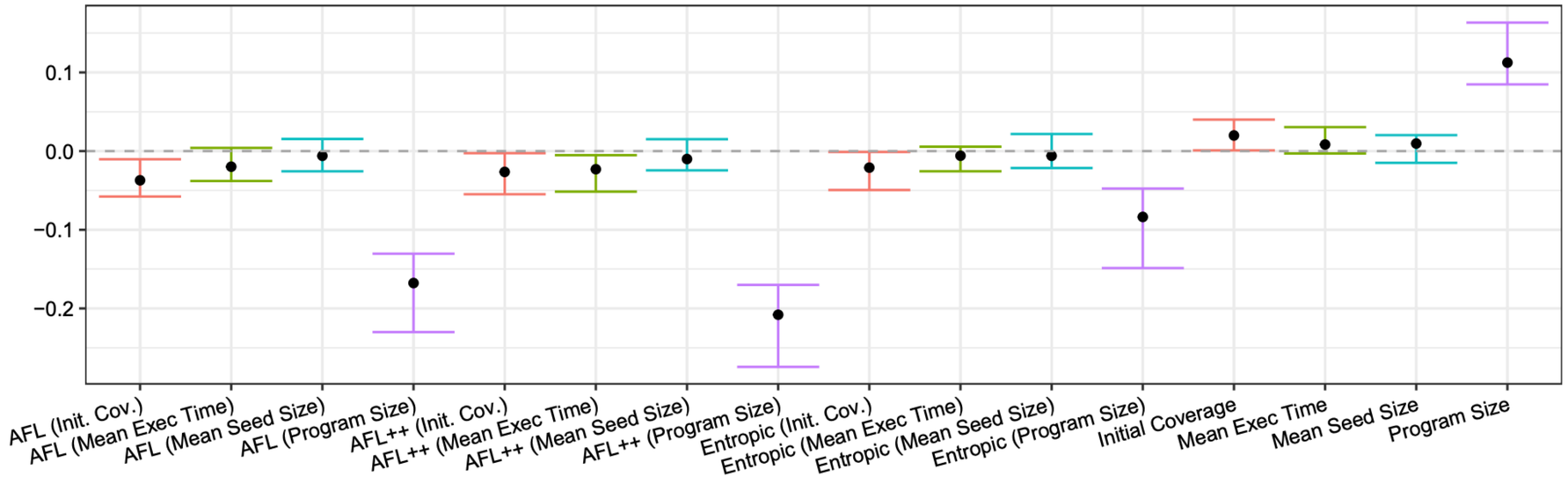


Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking ~ Fuzzer × Properties) [Eqn. 1].

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis

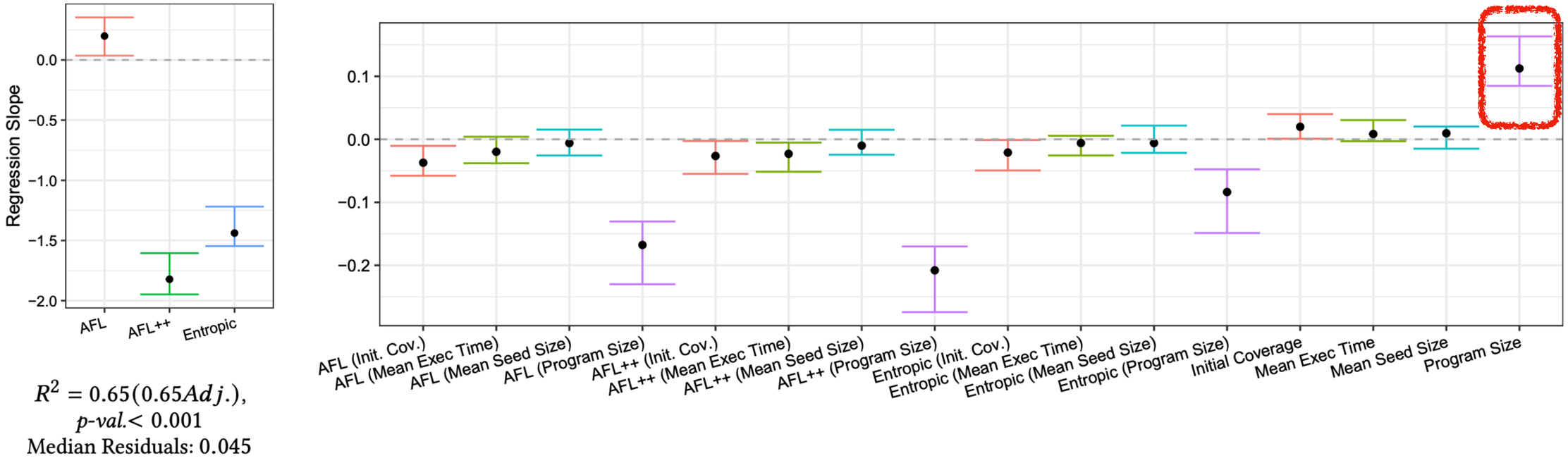
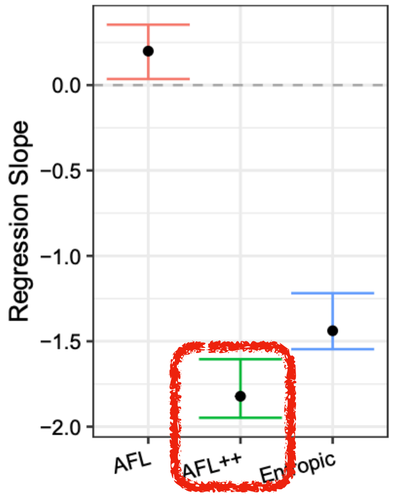


Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking ~ Fuzzer × Properties) [Eqn. 1].

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis



$R^2 = 0.65 (0.65 Adj.)$,
$p\text{-}val. < 0.001$
Median Residuals: 0.045

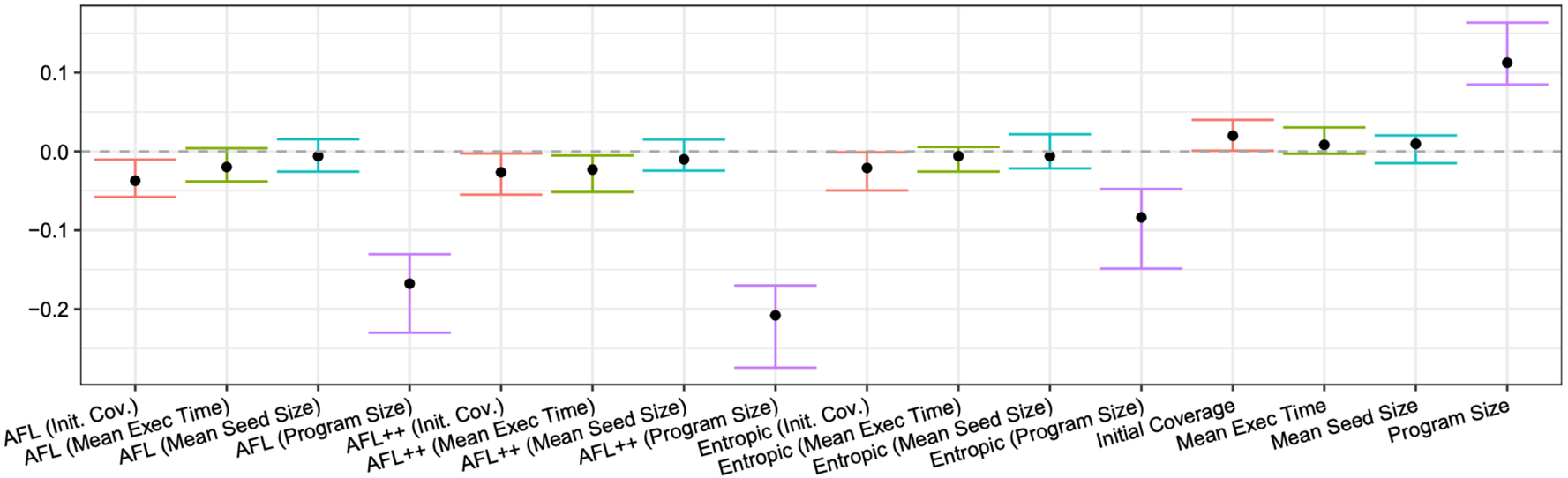


Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking ~ Fuzzer × Properties) [Eqn. 1].

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis

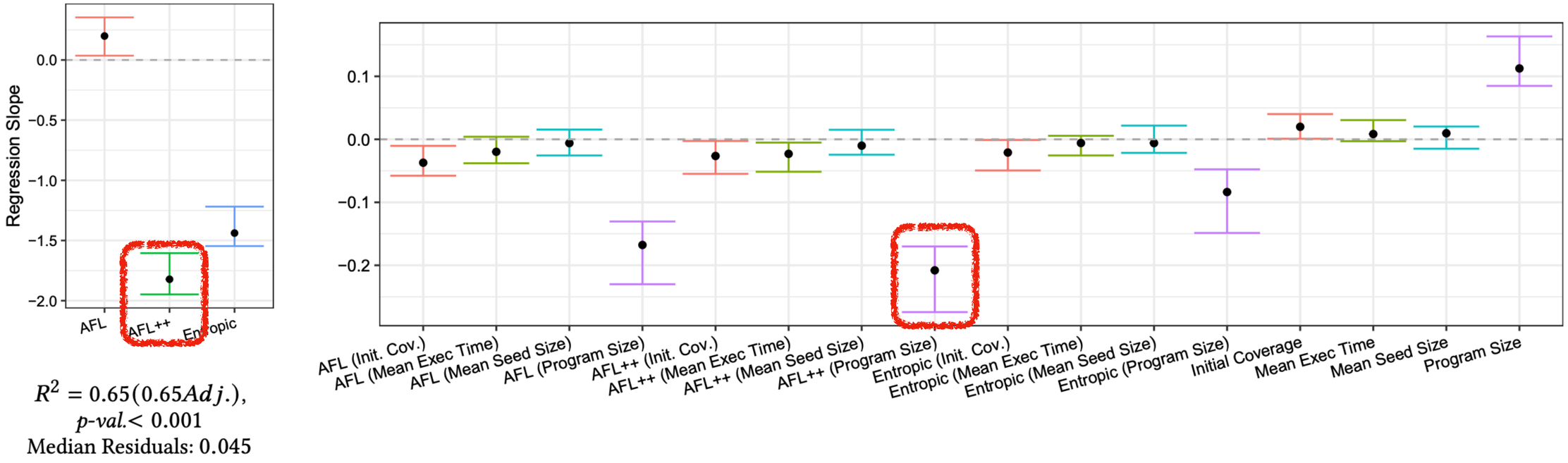


Fig. 5. Multiple Linear Regression with LibFuzzer as reference level (Fuzzer Ranking ~ Fuzzer × Properties) [Eqn. 1].

Benchmarks are specific, our claims general.

- We realize that the specific benchmark outcome is a function of the specific benchmark properties.
- We propose a counterfactual analysis

Benchmark Config 1 ↓ *Low Initial Coverage* ↓ *Small Programs* ↓ *Small and Fast Seeds*	**Benchmark Config 2** ↓ *Low Initial Coverage* ↑ *Large Programs* ↓ *Small and Fast Seeds*	**Benchmark Config 3** – *Median Initial Coverage* – *Median Sized Programs* – *Median Size and Speed Seeds*	**Benchmark Config 4** ↑ *High Initial Coverage* ↑ *Large Programs* ↑ *Large and Slow Seeds*	**Official FB Config** · *Static seed set used by Fuzzbench in all prior work*
1. Entropic LibFuzzer 3. AFL++ 4. AFL	1. AFL++ Entropic 3. AFL LibFuzzer	1. AFL++ 2. Entropic 3. LibFuzzer 4. AFL	1. AFL++ 2. Entropic AFL 4. LibFuzzer	1. / 2. AFL++ / Entropic 2. / 3. Entropic / AFL 4. LibFuzzer

Fig. 6. (left) Benchmarking outcomes at various levels of program and corpus properties (significant at bootstrapped 95% CI), (right) Benchmarking outcome from the Fuzzbench default corpora (significant at $p < 0.05$, Mann-Whitney U-test)

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark c
  - Techniques might seem to perform similar on the average ins
    Atomistic benchmarking hides the strengths of individual tec
- Recommendation:
  - Conduct a counterfactual analysis to report the conditions u
    benchmark outcome changes.

Dylan Wolff
NUS

Abhik Roychoudhury
NUS

Benchmarks are specific, our claims general.

TOSEM'25

Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome*. This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization*, can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

Dylan Wolff
NUS

Abhik Roychoudhury
NUS

Intermission. Any quick questions?

Goodhart's Law

Section I

Automated
Software Testing

Charles Goodhart on benchmarking outcome as a measure of progress.

Intermission. Any quick questions?

Goodhart's Law

Section II

Automated
Vulnerability Discovery

Benchmarking confirms **effectiveness**.
What about its **limits**?

Benchmarking confirms effectiveness. What about its limits?

Niklas Risse
MPI-SP

ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany

According to our survey of machine learning for vulnerability detection (ML4VD), 9 in every 10 papers published in the past five years define ML4VD as a function-level binary classification problem:

Given a function, does it contain a security flaw?

From our experience as security researchers, faced with deciding whether a given function makes the program vulnerable to attacks, we would often first want to understand the context in which this function is called.

In this paper, we study how often this decision can really be made without further context and study both vulnerable and non-vulnerable functions in the most popular ML4VD datasets. We call a function "*vulnerable*" if it was involved in a patch of an actual security flaw and confirmed to cause the program's vulnerability. It is "*non-vulnerable*" otherwise. We find that in almost all cases this decision *cannot* be made without further context. Vulnerable functions are often vulnerable only *because* a corresponding vulnerability-inducing calling context exists while non-vulnerable functions would often be vulnerable *if* a corresponding context existed.

But why do ML4VD techniques achieve high scores even though there is demonstrably not enough information in these samples? Spurious correlations: We find that high scores can be achieved even when only word counts are available. This shows that these datasets can be exploited to achieve high scores without actually detecting any security vulnerabilities.

We conclude that the prevailing problem statement of ML4VD is ill-defined and call into question the internal validity of this growing body of work. Constructively, we call for more effective benchmarking methodologies to evaluate the true capabilities of ML4VD, propose alternative problem statements, and examine broader implications for the evaluation of machine learning and programming analysis research.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: machine learning, vulnerability detection, benchmark, function, LLM, data quality, context, spurious correlations, ML4VD, software security

ACM Reference Format:
Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA018 (July 2025), 23 pages. https://doi.org/10.1145/3728887

1 Introduction

In recent years, the number of papers published on the topic of machine learning for vulnerability detection (ML4VD) has dramatically increased. Because of this rise in popularity, the validity and soundness of the underlying methodologies and datasets becomes increasingly important. So then, how exactly is the problem of ML4VD defined and thus evaluated?

ACM 2994-970X/2025/7-ARTISSTA018
https://doi.org/10.1145/3728887

USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Recent results of machine learning for automatic vulnerability detection (ML4VD) have been very promising. Given only the source code of a function f, ML4VD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate ML4VD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating ML4VD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of ML4VD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six ML4VD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art ML4VD techniques are unable to distinguish vulnerable functions from their patches.

1 Introduction

Recently several different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1, 12–15, 28]. The resulting models seem to outperform traditional program analysis methods, e.g. static analysis, even without requiring any hard-coded knowledge of program semantics or computational models. So, does this mean that the problem of detecting security vulnerabilities in software is solved? Are these models actually able to detect security vulnerabilities, or do the reported scores provide a false sense of security?

Even though ML4VD techniques achieve high scores on vulnerability detection benchmark datasets, there are still situations in which they fail to meet expectations when presented with new data. For example, it is possible to apply small semantic preserving changes to augment the testing dataset of a state-of-the-art model and then measure whether the model changes its predictions. If it does, it would indicate a dependence of the prediction on unrelated features. Examples of such transformations are identifier renaming [18,38,39,41,42], insertion of unexecuted statements [18, 35, 39, 41] or replacement of code elements with equivalent elements [2, 21]. The impact of augmenting testing data using these transformations has been explored for many different software-related tasks and the results seem to be clear: Learning-based models fail to perform well when testing data gets augmented using semantic preserving transformations of code [2, 5, 18, 30, 35, 38, 39, 41, 42].

In our own experiments, we were able to reproduce the findings of the literature and made additional observations: ML4VD techniques that were trained on typical training data for vulnerability detection are also unable to distinguish between vulnerable functions and their patched counterparts. If a patched function is also predicted as vulnerable, this indicates that the prediction critically depends on features unrelated to the presence of a security vulnerability.

It has previously been proposed to reduce the dependence on unrelated features by augmenting not just the testing data but also the training data [5, 18, 35, 38, 39, 41, 42]. Indeed, this seems to restore the lost performance back to previous levels, but does it really reduce the dependence on unrelated features, or are the models just overfitting to different unrelated features of the data?

In this paper, we propose a novel benchmarking methodology that can be used to evaluate the capabilities of ML4VD techniques by using data augmentation. First, we propose

FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

ABSTRACT

Recent results of machine learning for automatic vulnerability detection have been very promising indeed: Given only the source code of a function f, models trained by machine learning techniques can decide if f contains a security flaw with up to 70% accuracy.

But how do we know that these results are general and not specific to the datasets? To study this question, researchers proposed to *amplify* the testing set by injecting semantic preserving changes and found that the model's accuracy significantly drops. In other words, the model uses *some* unrelated features during classification. In order to increase the robustness of the model, researchers proposed to train on amplified training data, and indeed model accuracy increased to previous levels.

In this paper, we replicate and continue this investigation, and provide an actionable model benchmarking methodology to help researchers better evaluate advances in machine learning for vulnerability detection. Specifically, we propose a cross validation algorithm, where a semantic preserving transformation is applied during the amplification of either the training set or the testing set. Using 11 transformations and 3 ML techniques, we find that the improved robustness only applies to the specific transformations used during training data amplification. In other words, the robustified models still rely on unrelated features for predicting the vulnerabilities in the testing data.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

machine learning, automatic vulnerability detection, semantic preserving transformations, large language models

ACM Reference Format:
Niklas Risse. 2023. Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3611643.3617845

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3617845

1 INTRODUCTION

Recently a number of different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1, 5–8, 14]. So, does this mean that the problem of detecting security vulnerabilities in software is solved? How do we know that the reported results are general and not specific to the benchmark datasets?

To study these questions, researchers have tried to explore the capabilities and limits of machine learning techniques in ways that go beyond simple evaluations on benchmark testing sets. For example, it is possible to apply small semantic preserving amplifications to the input programs of a state-of-the-art model and then measure, whether the model changes its predictions and whether it still performs well. Examples for such amplifications are identifier renaming [9, 17–20], insertion of unexecuted statements [9, 16, 18, 19] or replacement of code elements with equivalent elements [3, 10]. The impact of applying semantic preserving amplifications to testing data has been explored for many different tasks in software engineering, and the results seems to be clear: Machine learning techniques lack robustness against semantic preserving amplifications [3, 4, 9, 11, 15–20].

A common strategy to address the robustness problem is training data amplification; applying the same or similar amplifications to the training dataset. Many of the works that reported the lack of robustness of ML models when trained on unamplified data also investigated training data amplification using their respective methods [4, 9, 11, 16–20]. They found a restoration or at least improvement towards the initial high performance. But does training data amplification actually improve the ability of these models to detect vulnerabilities, or are they just overfitting to a different set of data?

We contribute to answering this question by proposing a general benchmarking methodology that can be used to evaluate the capabilities of machine learning models for vulnerability detection by using data amplification. The core of the methodology is a cross validation, in which a selected semantic preserving amplification method is applied to the training dataset of a model, and a *different* amplification method is applied to the testing dataset (see Figure 1). When repeated for all possible pairs out of a set of amplification methods, the resulting scores provide a measure of overfitting to the specific semantic preserving amplification methods that were used during training data amplification.

In addition to the general methodology, we present the results of an empirical study, in which we apply the proposed methodology to three state-of-the-art ML techniques for vulnerability detection. We implemented 11 different semantic preserving amplification methods and tried to cover types of amplifications commonly used

Benchmarking confirms effectiveness. What about its limits?

Table 2: Classification accuracies and F1 scores in percentages: The two far-right columns give the maximum and average relative difference in accuracy/F1 compared to [redacted] model with the composite code representations, i.e., [redacted](Composite).

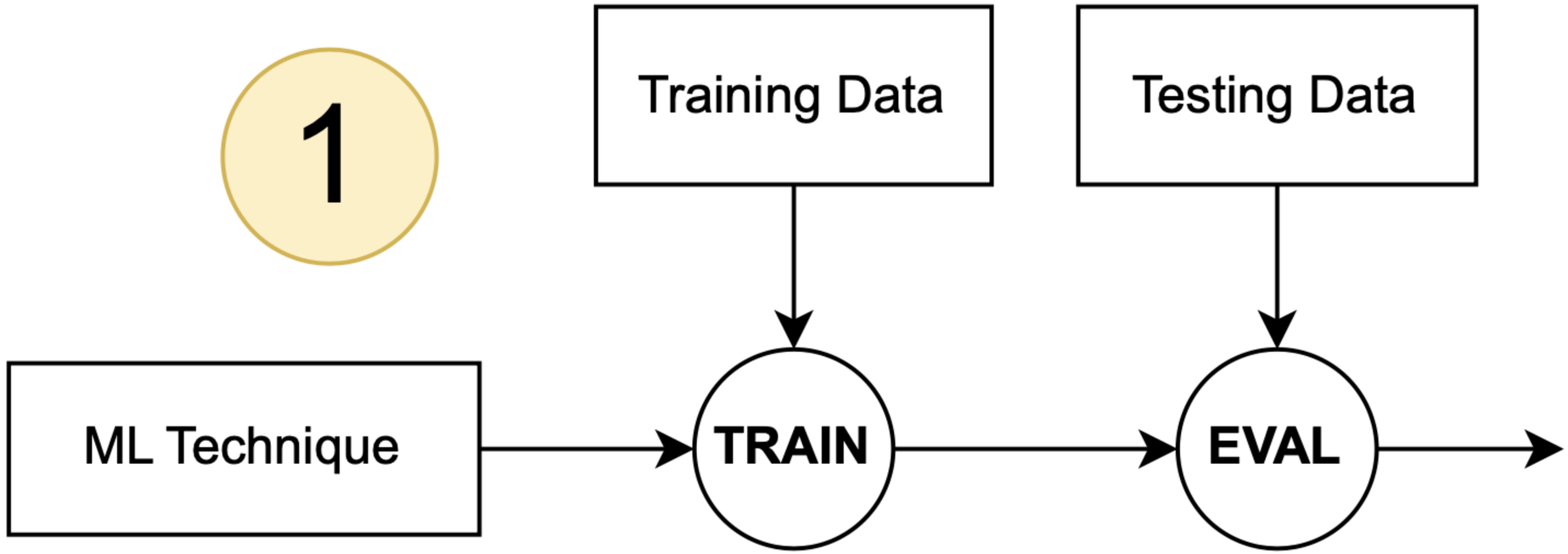
Method	Linux Kernel		QEMU		Wireshark		FFmpeg		Combined		Max Diff		Avg Diff	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1	ACC	F1
Metrics + Xgboost	67.17	79.14	59.49	61.27	70.39	61.31	67.17	63.76	61.36	63.76	14.84	11.80	10.30	8.71
3-layer BiLSTM	67.25	80.41	57.85	57.75	69.08	55.61	53.27	69.51	59.40	65.62	16.48	15.32	14.04	8.78
3-layer BiLSTM + Att	75.63	82.66	65.79	59.92	74.50	58.52	61.71	66.01	69.57	68.65	8.54	13.15	5.97	7.41
CNN	70.72	79.55	60.47	59.29	70.48	58.15	53.42	66.58	63.36	60.13	16.16	13.78	11.72	9.82
[redacted] *n* (AST)	72.65	81.28	70.08	66.84	79.62	64.56	63.54	70.43	67.74	64.67	6.93	8.59	4.69	5.01
[redacted] *n* (CFG)	78.79	82.35	71.42	67.74	79.36	65.40	65.00	71.79	70.62	70.86	4.58	5.33	2.38	2.93
[redacted] *n* (NCS)	78.68	81.84	72.99	69.98	78.13	59.80	65.63	69.09	70.43	69.86	3.95	8.16	2.24	4.45
[redacted] (DFG_C)	70.53	81.03	69.30	56.06	73.17	50.83	63.75	69.44	65.52	64.57	9.05	17.13	6.96	10.18
[redacted] (DFG_R)	72.43	80.39	68.63	56.35	74.15	52.25	63.75	71.49	66.74	62.91	7.17	16.72	6.27	9.88
[redacted] DFG_W)	71.09	81.27	71.65	65.88	72.72	51.04	64.37	70.52	63.05	63.26	9.21	16.92	6.84	8.17
[redacted] Composite)	74.55	79.93	72.77	66.25	78.79	67.32	64.46	70.33	70.35	69.37	5.12	6.82	3.23	3.92
[redacted] (AST)	**80.24**	84.57	71.31	65.19	79.04	64.37	65.63	71.83	69.21	69.99	3.95	7.88	2.33	3.37
[redacted] (CFG)	80.03	82.91	74.22	70.73	79.62	66.05	66.89	70.22	71.32	71.27	2.69	3.33	1.00	2.33
[redacted] (NCS)	79.58	81.41	72.32	68.98	79.75	65.88	67.29	68.89	70.82	68.45	2.29	4.81	1.46	3.84
[redacted] DFG_C)	78.81	83.87	72.30	70.62	79.95	66.47	65.83	70.12	69.88	70.21	3.75	3.43	2.06	2.30
[redacted] DFG_R)	78.25	80.33	73.77	70.60	80.66	66.17	66.46	72.12	71.49	70.92	3.12	4.64	1.29	2.53
[redacted] DFG_W)	78.70	84.21	72.54	71.08	80.59	66.68	67.50	70.86	71.41	71.14	2.08	2.69	1.27	1.77
[redacted] omposite)	79.58	**84.97**	**74.33**	**73.07**	**81.32**	**67.96**	**69.58**	**73.55**	**72.26**	**73.26**	-	-	-	-

Benchmarking confirms effectiveness. What about its limits?

- Experimental Setup:
  - Dataset: CodeXGLUE/Devign (45.6% vulnerable functions).
  - 6 SOTA ML4VD approaches (mostly LLMs, one graph-based).

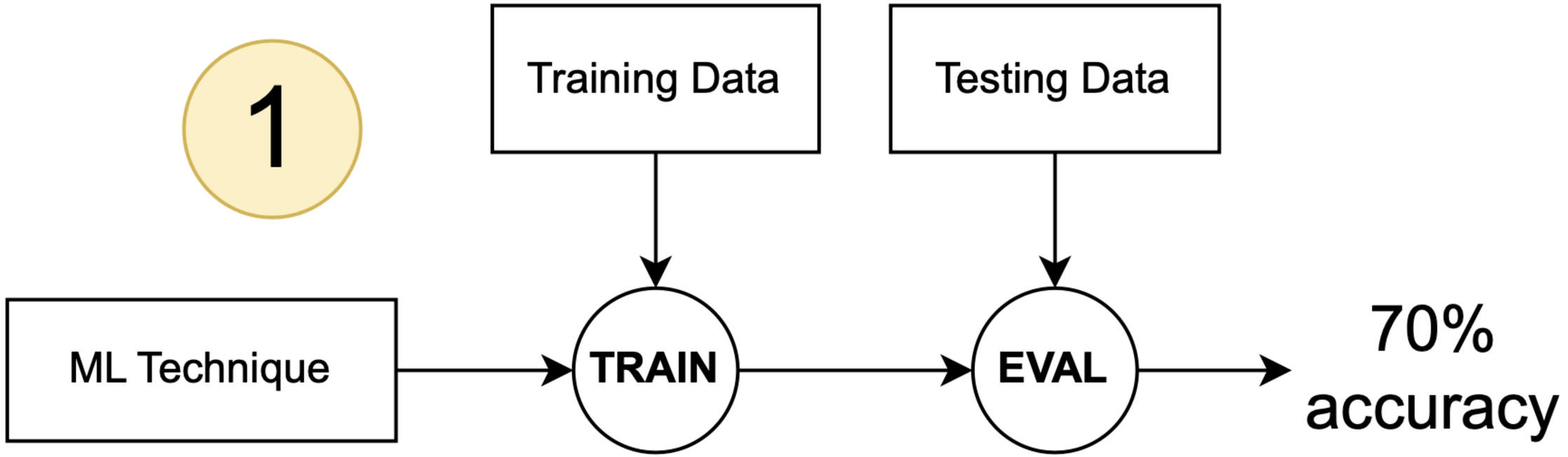
Benchmarking confirms effectiveness. What about its limits?

- ML4VD



Benchmarking confirms effectiveness. What about its limits?

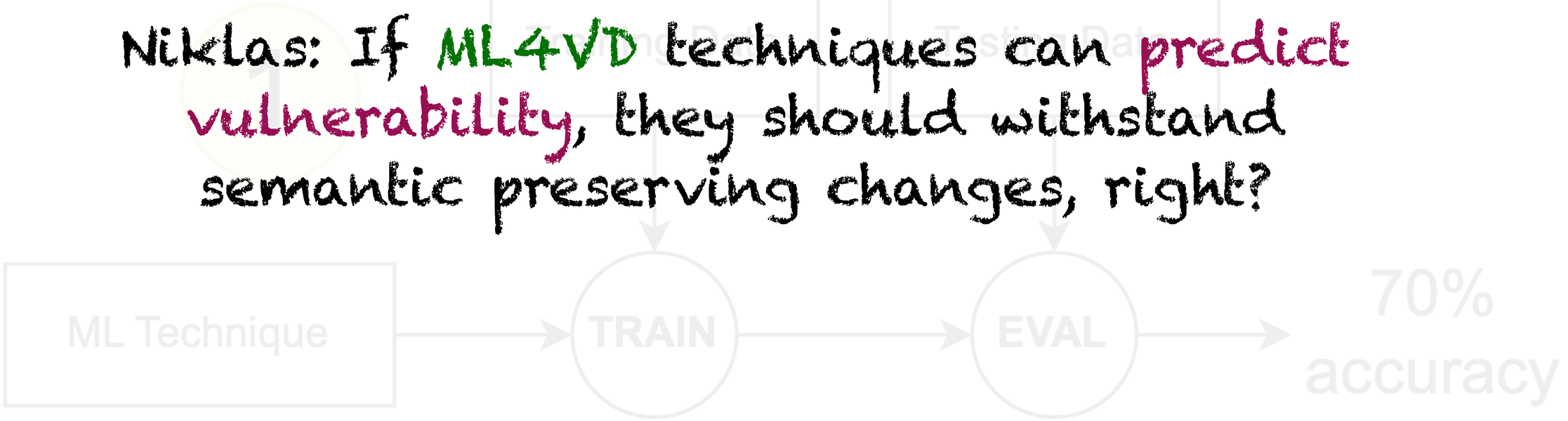
- ML4VD



Benchmarking confirms effectiveness. What about its limits?

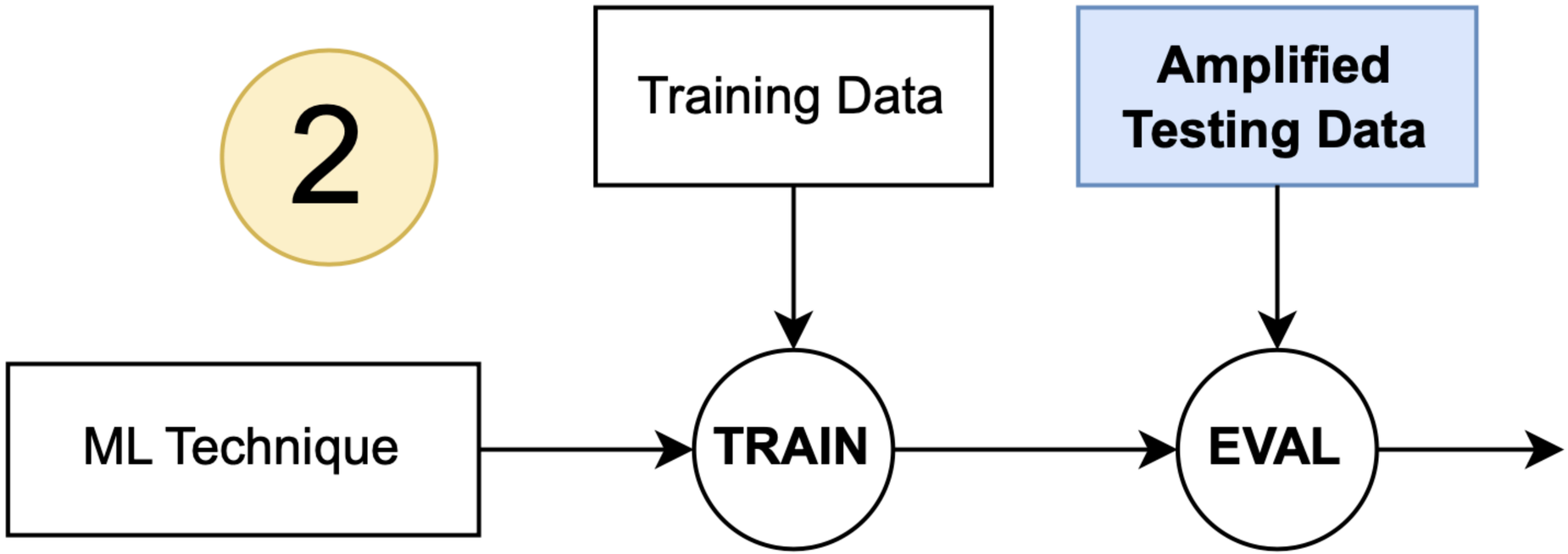
- ML4VD

Niklas: If ML4VD techniques can predict vulnerability, they should withstand semantic preserving changes, right?



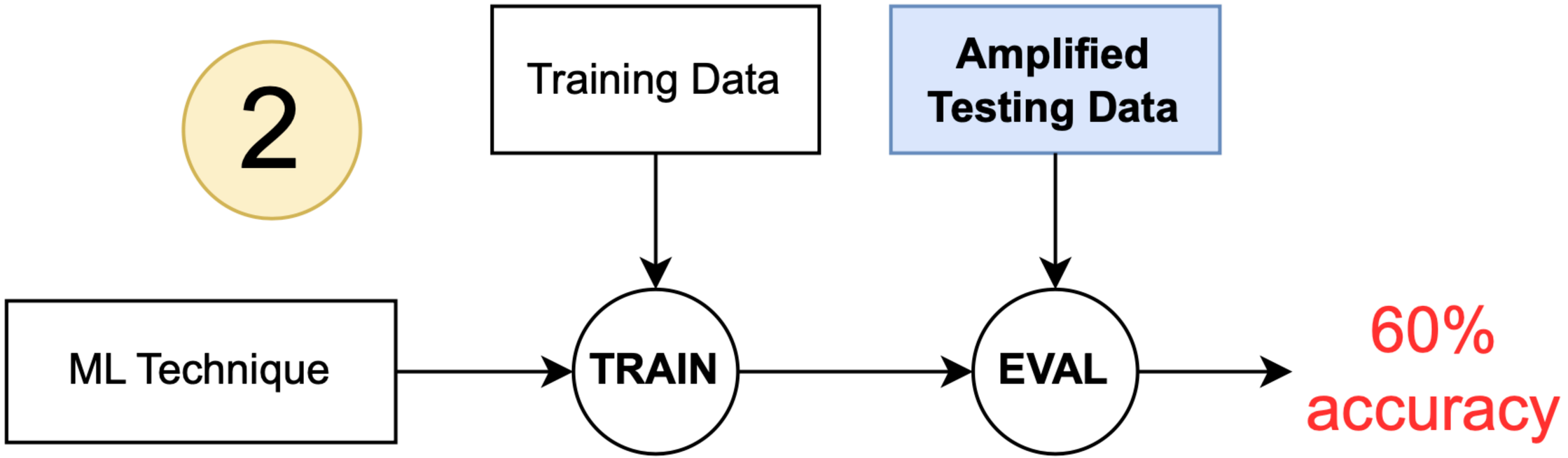
Benchmarking confirms effectiveness. What about its limits?

- ML4VD — but testing is amplified with semantic-preserving changes



Benchmarking confirms effectiveness. What about its limits?

- ML4VD — but testing is amplified with semantic-preserving changes



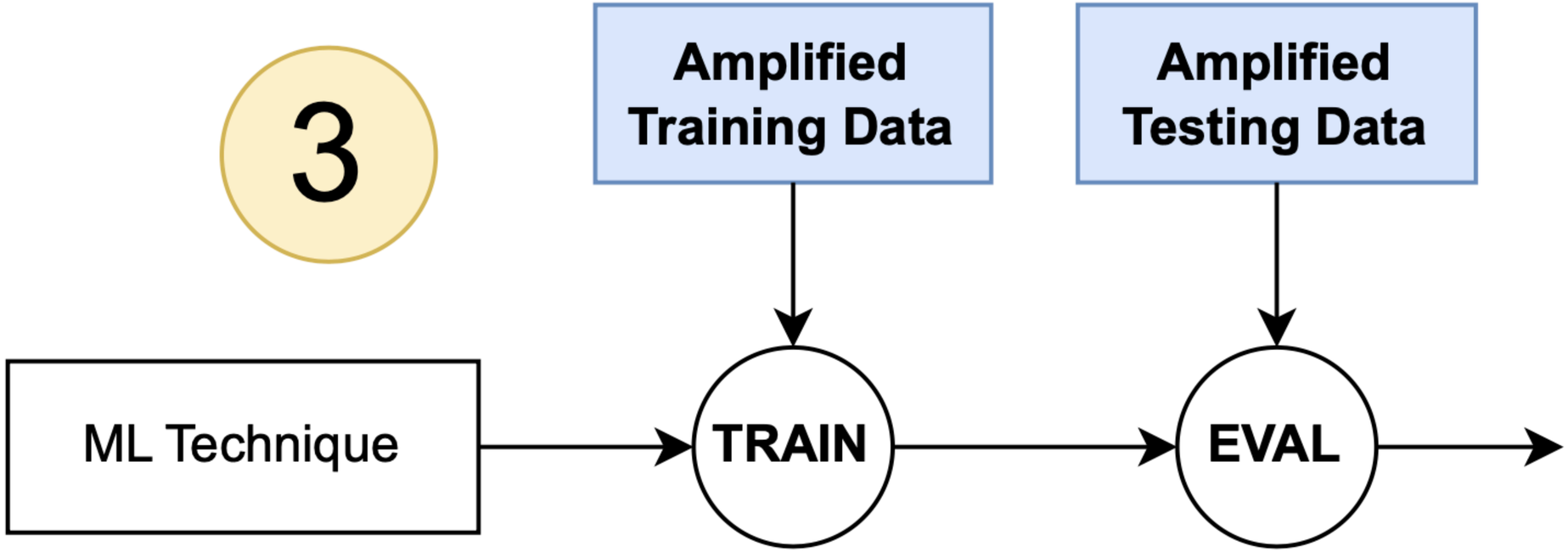
Benchmarking confirms effectiveness. What about its limits?

- ML4VD — but testing is amplified with semantic-preserving changes

Test failed: Not the semantic cause of the vulnerability, but something else makes it predict the vulnerability label correctly.

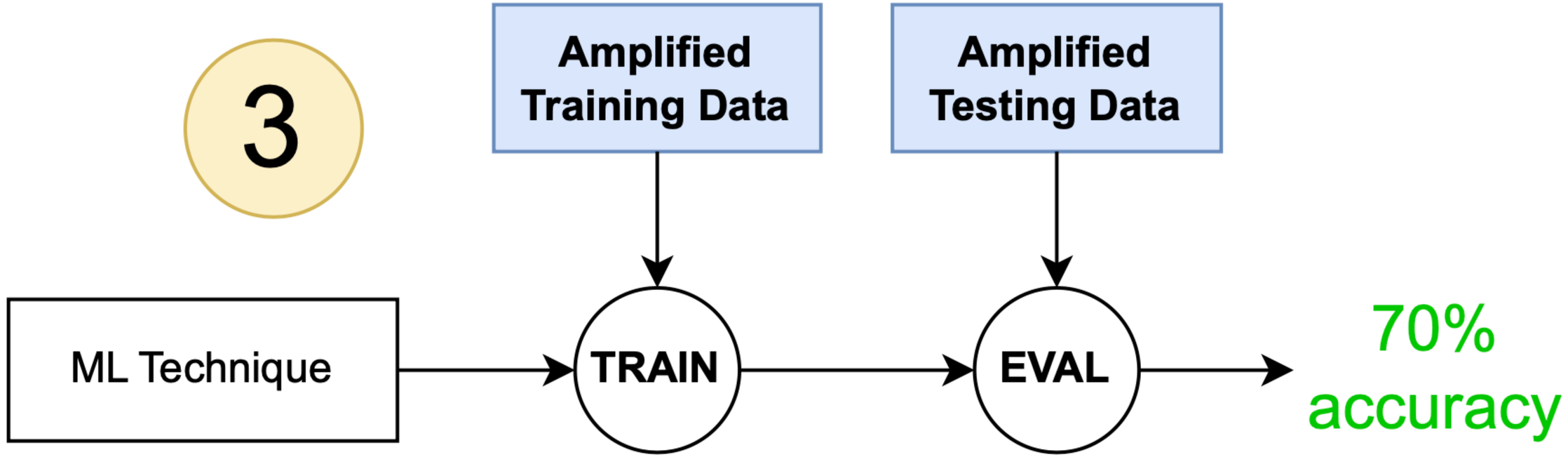
Benchmarking confirms effectiveness. What about its limits?

- ML4VD — Robustified



Benchmarking confirms effectiveness. What about its limits?

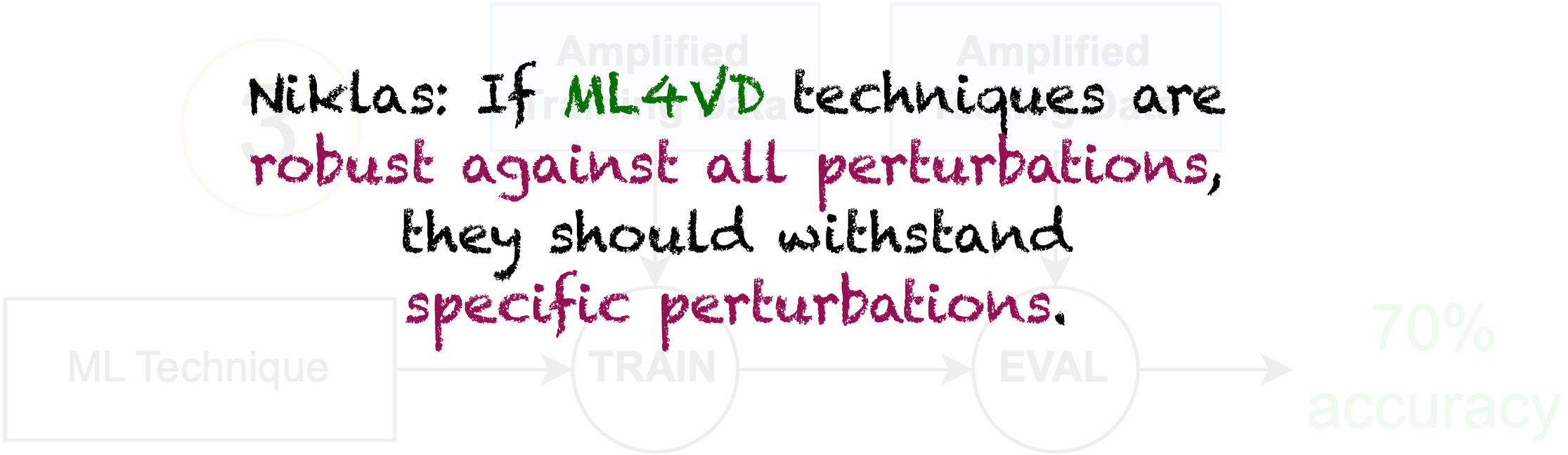
- ML4VD — Robustified



Benchmarking confirms effectiveness. What about its limits?

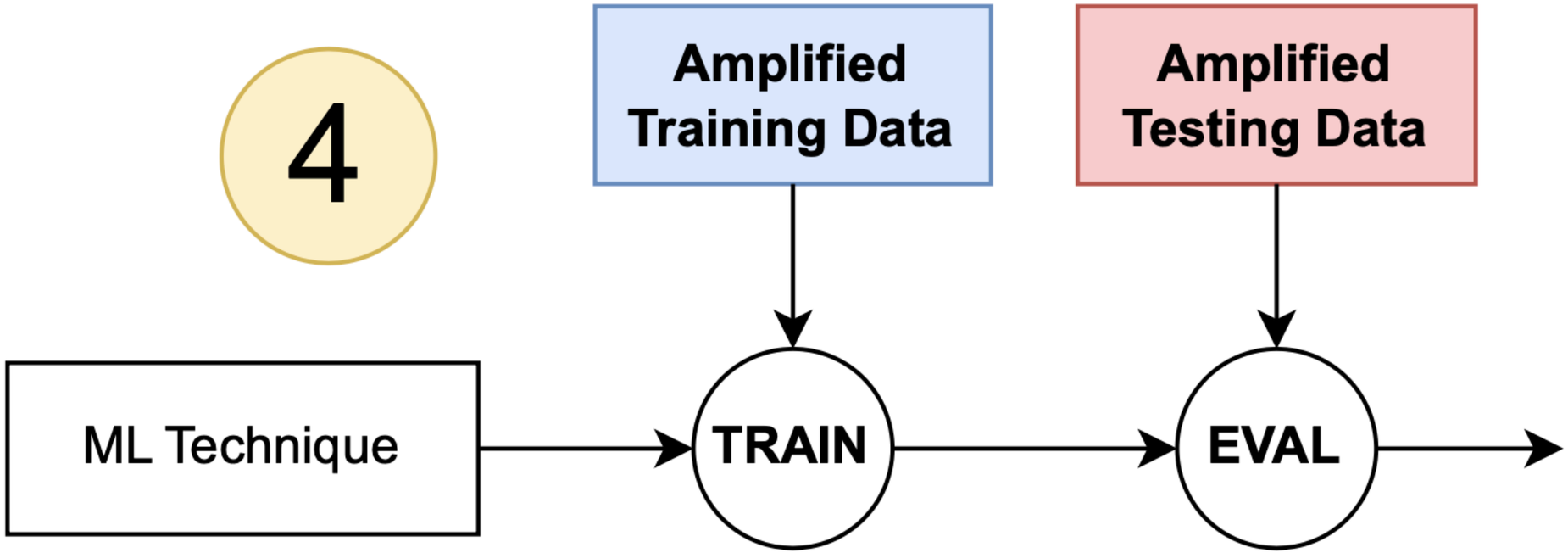
- ML4VD — Robustified

Niklas: If ML4VD techniques are robust against all perturbations, they should withstand specific perturbations.



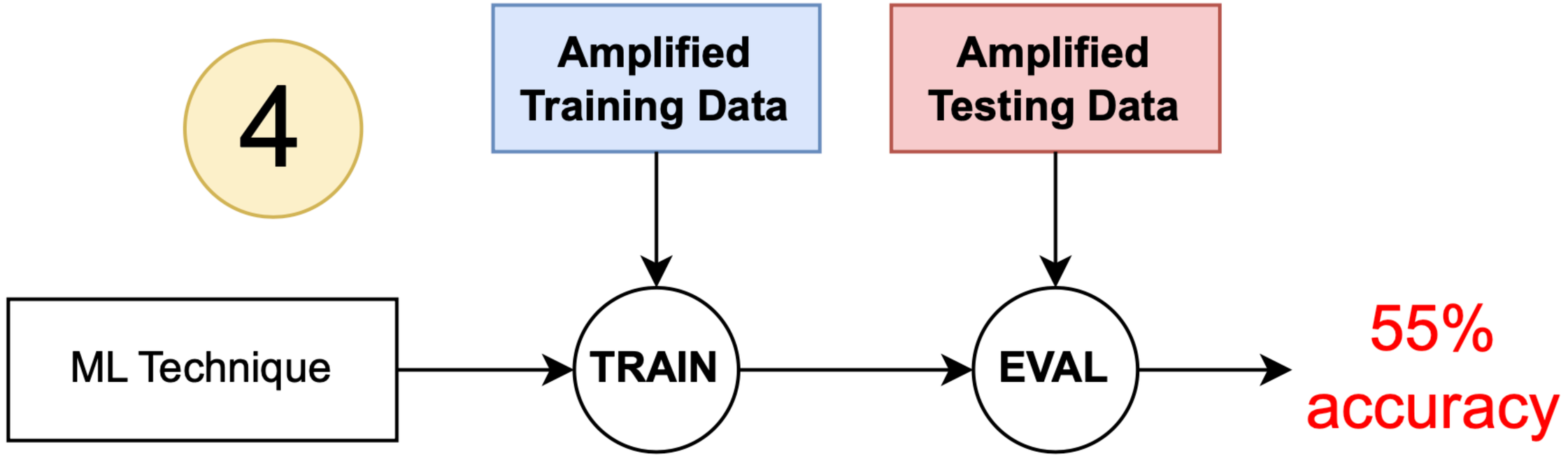
Benchmarking confirms effectiveness. What about its limits?

- ML4VD — Robustified + testing is amplified *hold-one-out*



Benchmarking confirms effectiveness. What about its limits?

- ML4VD — Robustified + testing is amplified *hold-one-out*



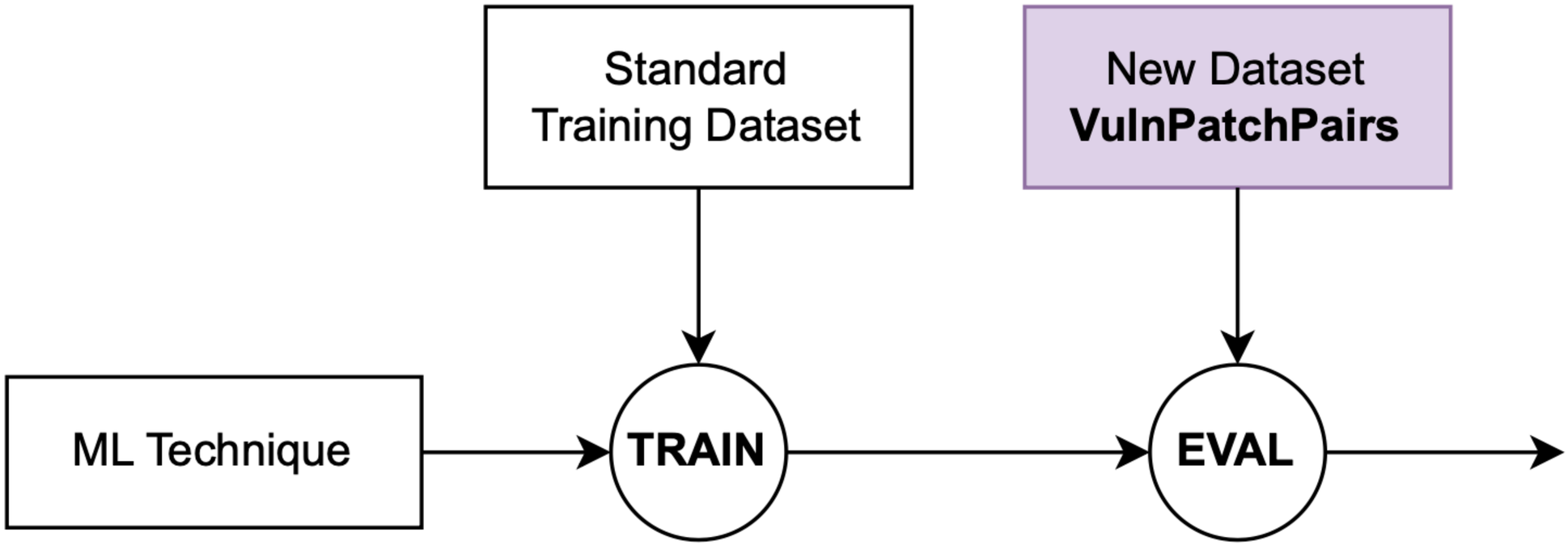
Benchmarking confirms effectiveness. What about its limits?

- ML4VD — Robustified + testing is amplified *hold-one-out*

Test failed: The model now only overfits to the specific way in which we robustify the model. There is still an alternative explanation of the impressive results.

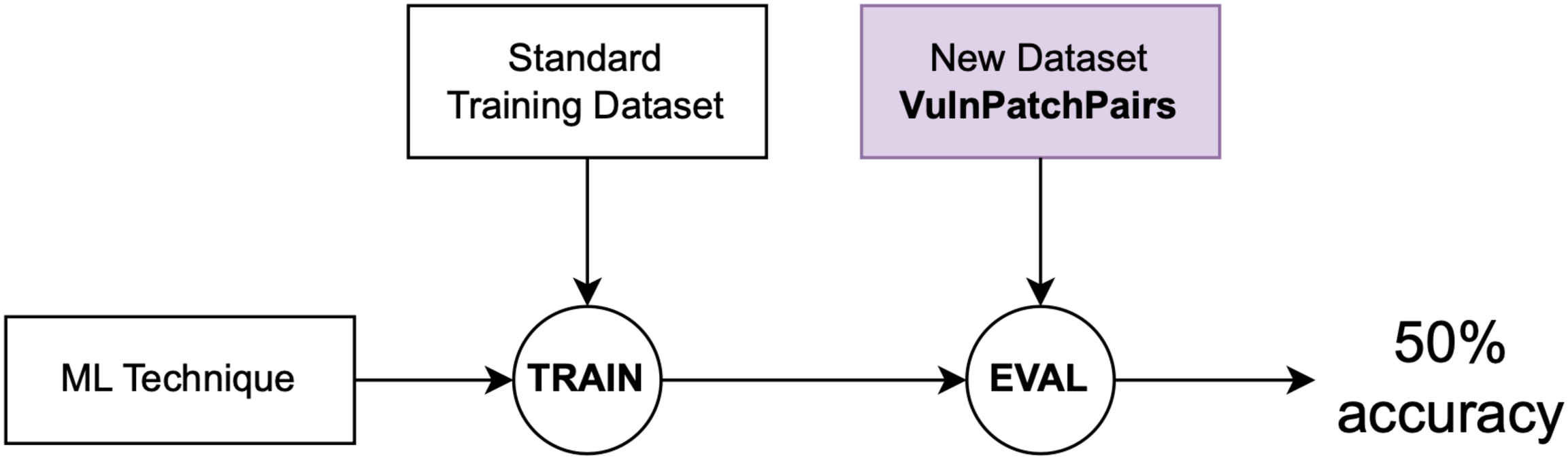
Benchmarking confirms effectiveness. What about its limits?

- Bonus: How well can it distinguish vulnerable and patched function?



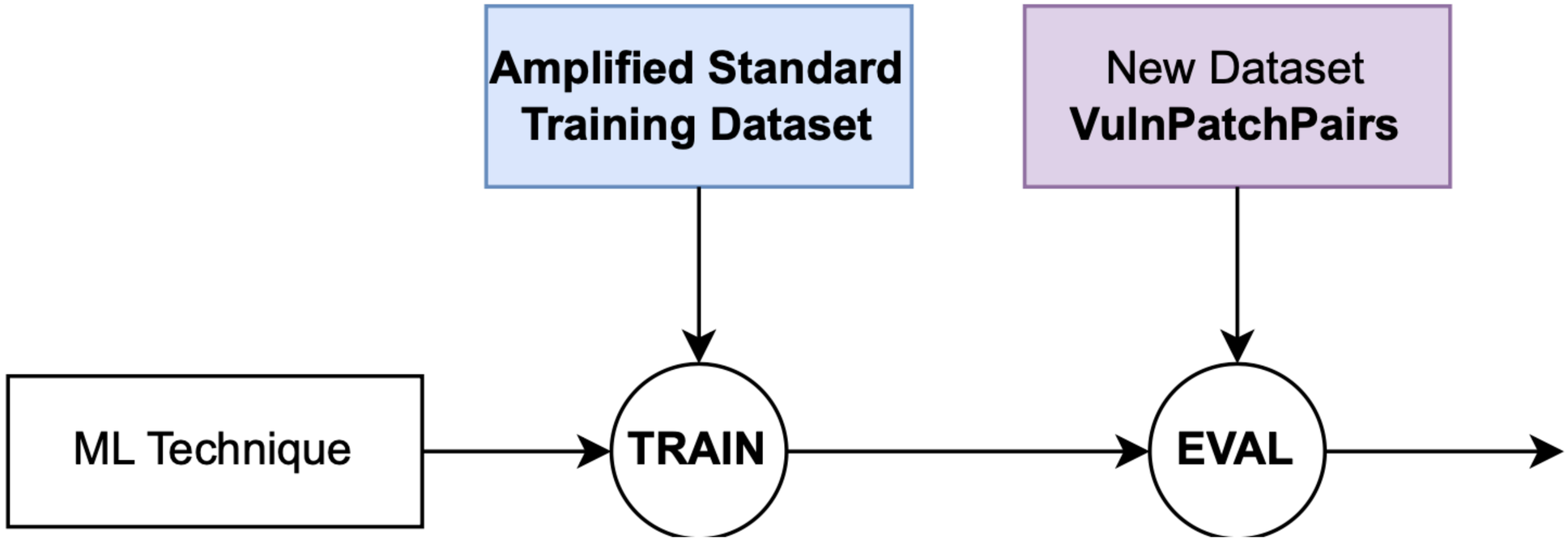
Benchmarking confirms effectiveness. What about its limits?

- Bonus: How well can it distinguish vulnerable and patched function?



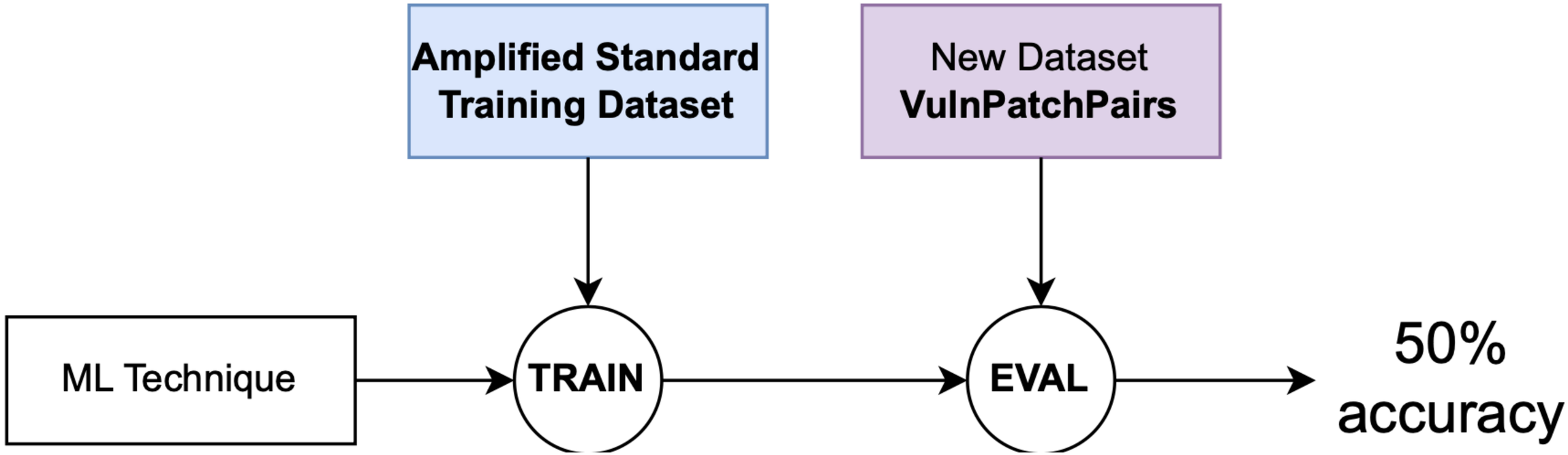
Benchmarking confirms effectiveness. What about its limits?

- Bonus: How well can it distinguish vulnerable and patched function?



Benchmarking confirms effectiveness. What about its limits?

- Bonus: How well can it distinguish vulnerable and patched function?



Benchmarking confirms effectiveness. What about its limits?

- Given such impressive results, the experimenter might assume they are explained by ML4VD's true capability to detect vulnerabilities.

- We study the veracity of this assumption:
  - Even simple semantic-preserving changes reduces observed effectiveness.
  - Making the model more robust doesn't change this insight.
  - ML4VD cannot even distinguish buggy from patched version.

- Alternative explanation of impressive results:
  - Spurious correlations with unrelated features.

Benchmarking confirms effectiveness. What about its limits?

- Given such impressive results, the experimenter might assume they are explained by ML4VD's true capability to detect vulnerabilities.
- We study the veracity of this assumption:
  - Even simple semantic-preserving changes reduces observed effectiveness.
  - Making the model more robust doesn't change this insight.
  - ML4VD cannot even distinguish buggy from patched version.
- Alternative explanation of impressive results:
  - Spurious correlations with unrelated features.

Benchmarking confirms effectiveness. What about its limits?

- Given such impressive results, the experimenter might assume they are explained by ML4VD's true capability to detect vulnerabilities.
- We study the veracity of this assumption:
  - Even simple semantic-preserving changes reduces observed effectiveness.
  - Making the model more robust doesn't change this insight.
  - ML4VD cannot even distinguish buggy from patched version.
- Alternative explanation of impressive results:
  - Spurious correlations with unrelated features.

Top Score on the Wrong Exam

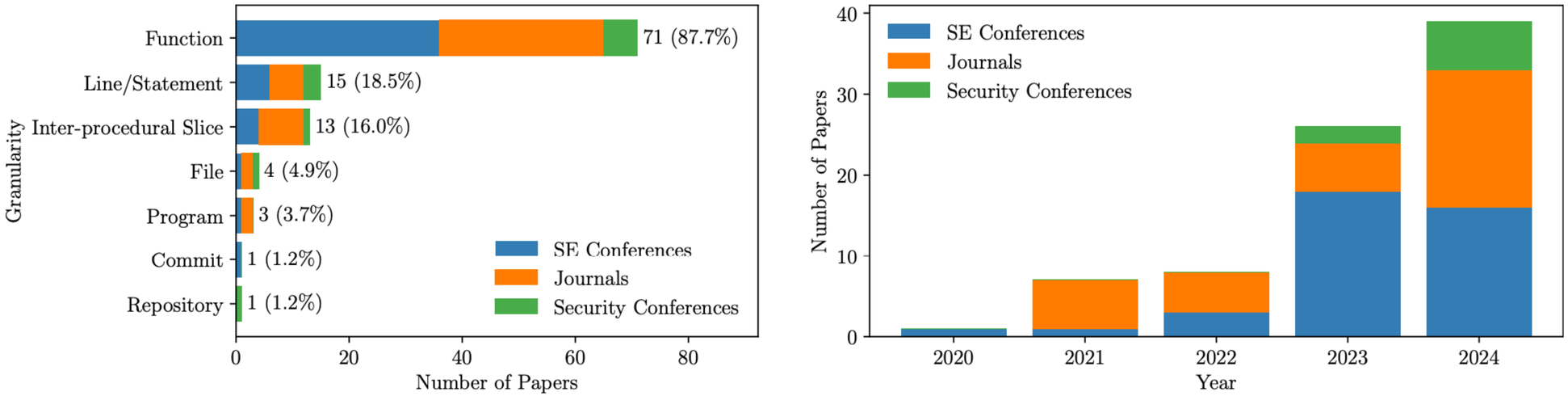
"Given this function, does it contain a security flaw?"

Does this question make sense?

What is the definition of security flaw? What does it even mean to contain a security flaw?
Let's say, the code in the function causes the program to be vulnerable to attack. Changing that function fixes the vulnerability.
Can we say whether the function causes the program to be vulnable without further context?
This is the question we study.

Top Score on the Wrong Exam

- ML4VD is mostly cast as binary classification problem.
  - “Given this function, does it contain a security flaw?”



(a) ML4VD papers per problem statement granularity.

(b) ML4VD papers per year.

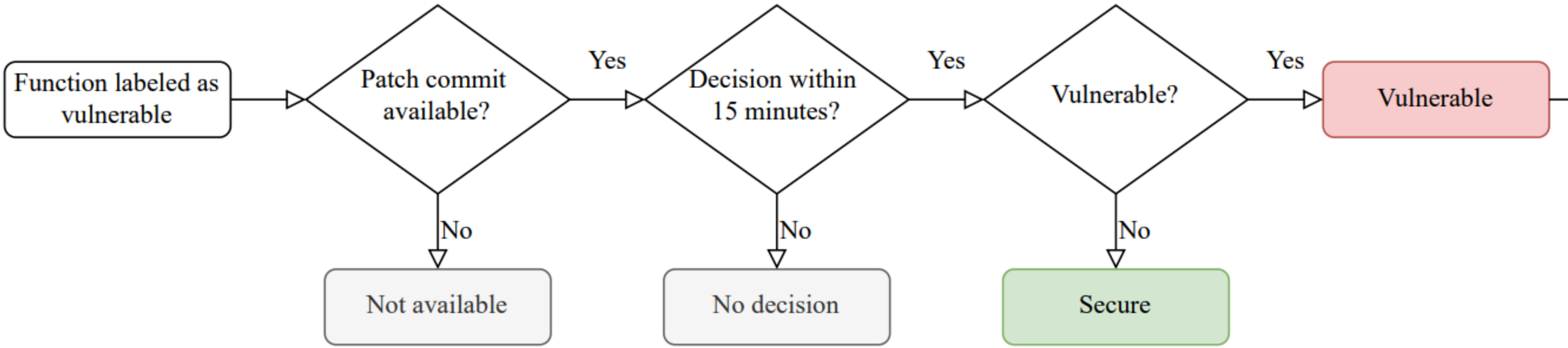
Fig. 2. Literature survey results for the 81 ML4VD papers we identified in the top Software Engineering (SE) and Security conferences and journals. Figure 2a shows how the papers define the problem of ML4VD. Note that a paper may use multiple granularities, which explains why the numbers in Figure 2a do not add up to 100%. Figure 2b shows how many papers were published each year since 2020.

Top Score on the Wrong Exam

```
1  TfLiteStatus ResizeOutputTensors(TfLiteContext* context, TfLiteNode* node,
2                                   const TfLiteTensor* axis,
3                                   const TfLiteTensor* input, int num_splits) {
4    int axis_value = GetTensorData<int>(axis)[0];
5    // [...]
6    const int input_size = SizeOfDimension(input, axis_value);
7    TF_LITE_ENSURE_MSG(context, input_size % num_splits == 0,
8                       "Not an even split");
9    const int slice_size = input_size / num_splits;
10   for (int i = 0; i < NumOutputs(node); ++i) {
11     TfLiteIntArray* output_dims = TfLiteIntArrayCopy(input->dims);
12     output_dims->data[axis_value] = slice_size;
13     // [...]
14     TF_LITE_ENSURE_STATUS(context->ResizeTensor(context, output, output_dims));
15   }
16   return kTfLiteOk;
17 }
```

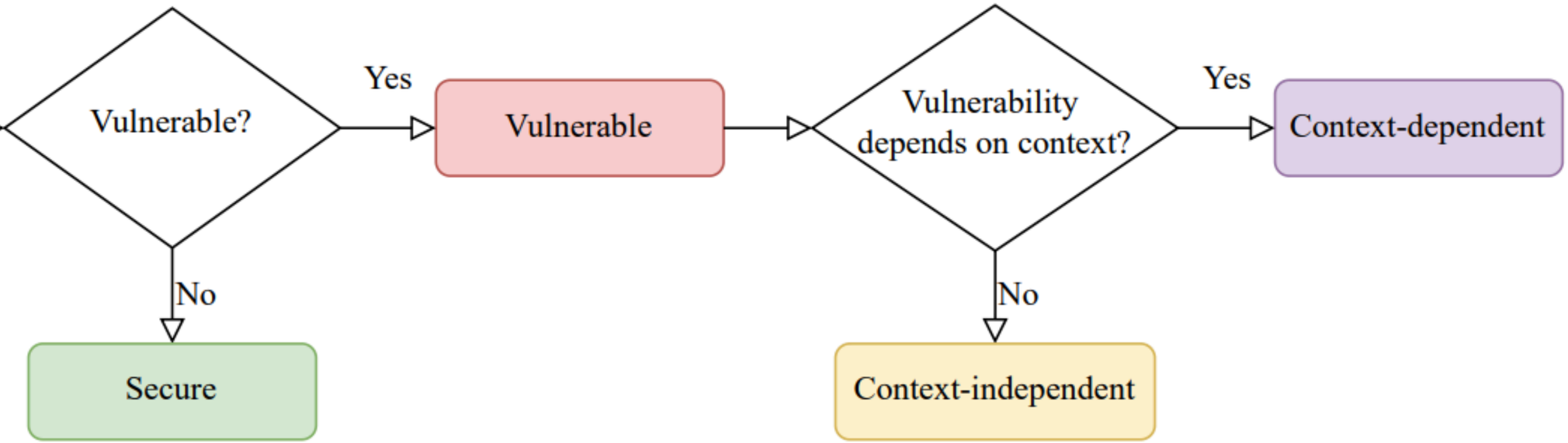
Fig. 1. Context-dependent vulnerability (CVE-2021-29599) in DiverseVul dataset. If the function is called with `num_splits=0`, it crashes with a division-by-zero in Line 7.

Top Score on the Wrong Exam



3x100 functions labeled as **`vulnerable`** sampled from the three most popular benchmarks (BigVul, Devign, DiverseVul)

Top Score on the Wrong Exam



3x100 functions labeled as **`vulnerable`** sampled from the three most popular benchmarks (BigVul, Devign, DiverseVul)

Top Score on the Wrong Exam

Context-dependency of functions labeled as vulnerable.

How often do we have to abstain when deciding without further context whether a given function (that is labeled as **`vulnerable`**) really causes the program to be vulnerable?

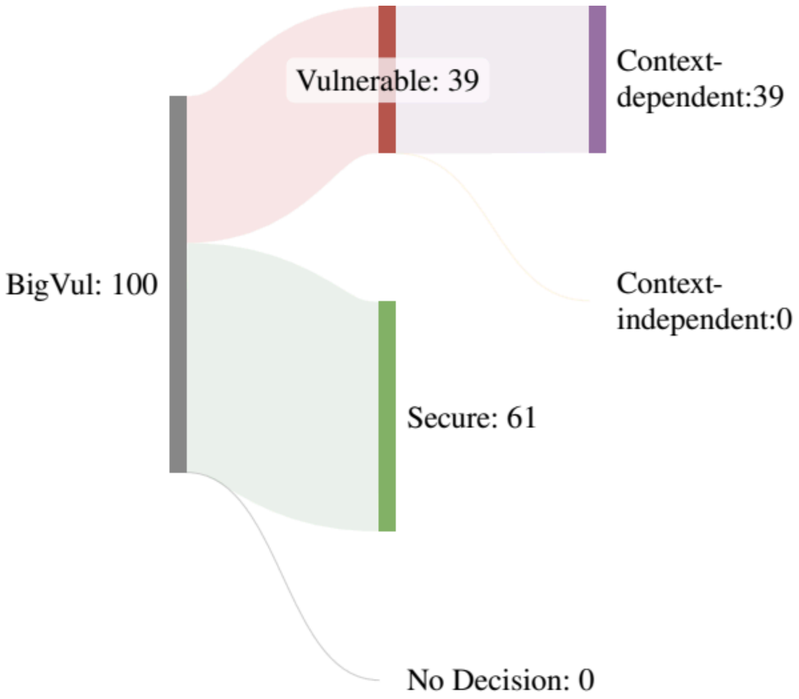
Top Score on the Wrong Exam

Context-dependency of functions labeled as vulnerable.

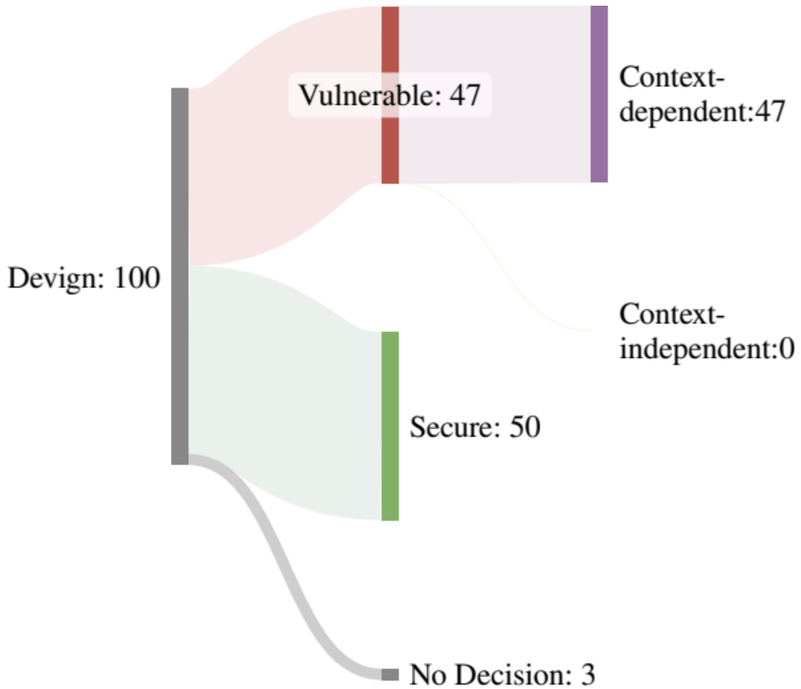
How often do we have to abstain when deciding without further context whether a given function (that is labeled as **`vulnerable`**) really causes the program to be vulnerable?

100% of the randomly sampled **`vulnerable`** functions.

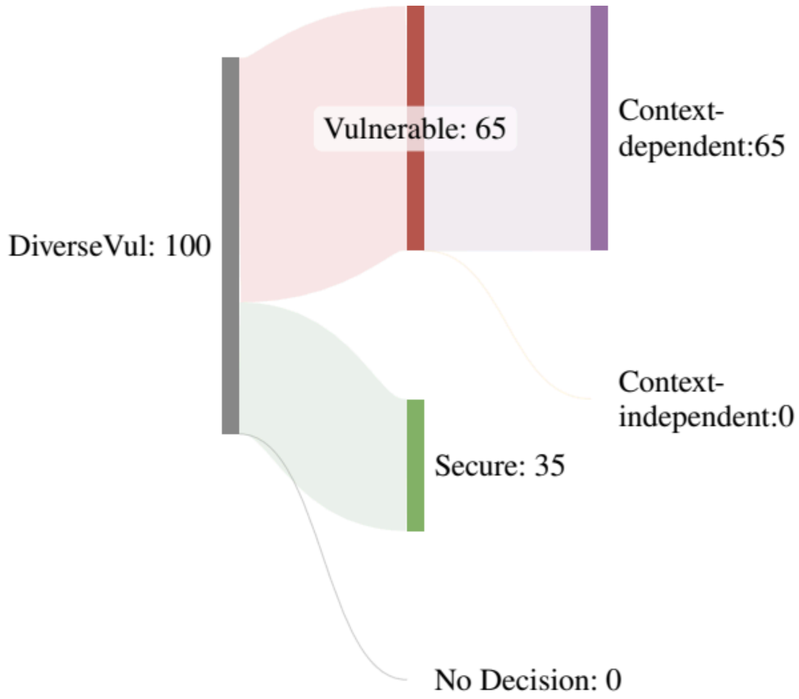
Top Score on the Wrong Exam



(a) **BigVul**



(b) **Devign**



(c) **DiverseVul**

Dataset	Function Argument	External Function	Type Declaration	Globals	Execution Environment
BigVul	16 (41%)	19 (49%)	1 (2%)	3 (8%)	0 (0%)
Devign	22 (47%)	20 (43%)	0 (0%)	5 (10%)	0 (0%)
DiverseVul	26 (40%)	34 (52%)	2 (3%)	1 (2%)	2 (3%)

Top Score on the Wrong Exam

What about functions that were labeled as secure?

How often would they make a program vulnerable IF a corresponding context existed?

Top Score on the Wrong Exam

What about functions that were labeled as secure?

How often would they make a program vulnerable IF a corresponding context existed?

92% of the randomly sampled **`secure`** functions.

Top Score on the Wrong Exam

- ML4VD as function-level, binary classification problem is ill-defined!
- Yet, ML4VD techniques perform impressively on these benchmarks.

Top Score on the Wrong Exam

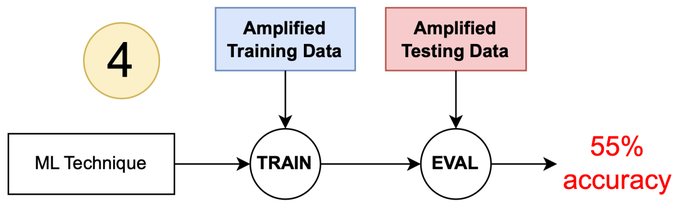
- ML4VD as function-level, binary classification problem is ill-defined!
- Yet, ML4VD techniques perform impressively on these benchmarks.

Why?

Hint:

Benchmarking confirms effectiveness.
What about its limits?

- ML4VD — Robustified + testing is amplified *hold-one-out*



Top Score on the Wrong Exam

- ML4VD as function-level, binary classification problem is ill-defined!
- Yet, ML4VD techniques perform impressively on these benchmarks.
- Why? Spurious correlations with features unrelated to vulnerability.
  - Even removing all information about vulnerability from functions, i.e., just using token counts, we get:

Top Score on the Wrong Exam

- ML4VD as function-level, binary classification problem is ill-defined!
- Yet, ML4VD techniques perform impressively on these benchmarks.
- Why? Spurious correlations with features unrelated to vulnerability.
  - Even removing all information about vulnerability from functions, i.e., just using token counts, we get:

62% f1-score on Devign.

86% f1-score on BigVul.

Top Score on the Wrong Exam

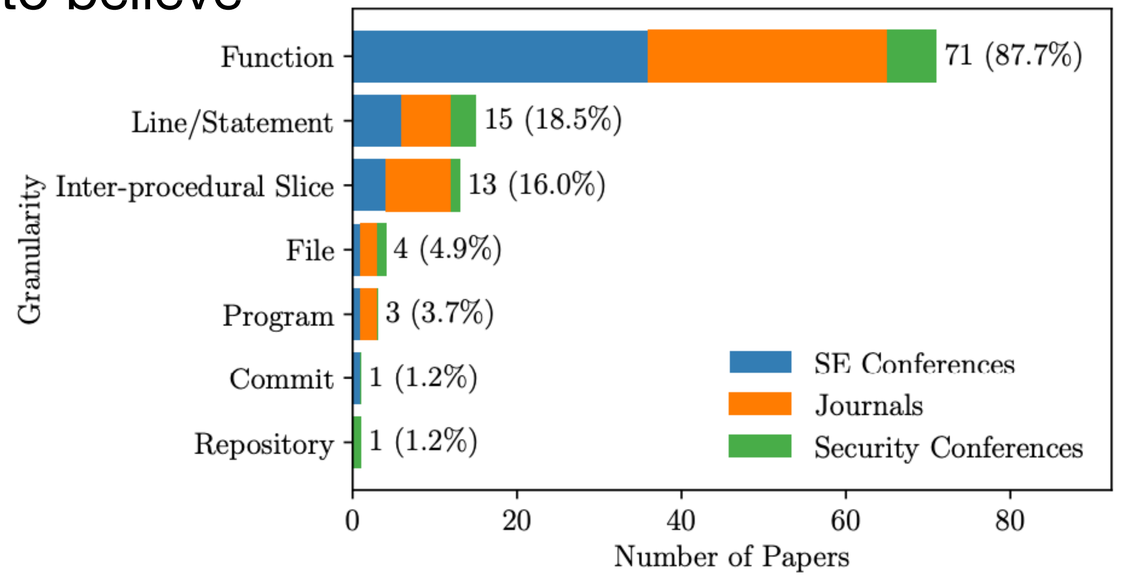
- What about alternative problem statements?

Top Score on the Wrong Exam

- What about alternative problem statements?
  - Classification with abstention.
    - Either classify into **`vulnerable`** / **`not vulnerable`** OR abstain entirely.
    - Impractical: Classifier would abstain in most cases.

Top Score on the Wrong Exam

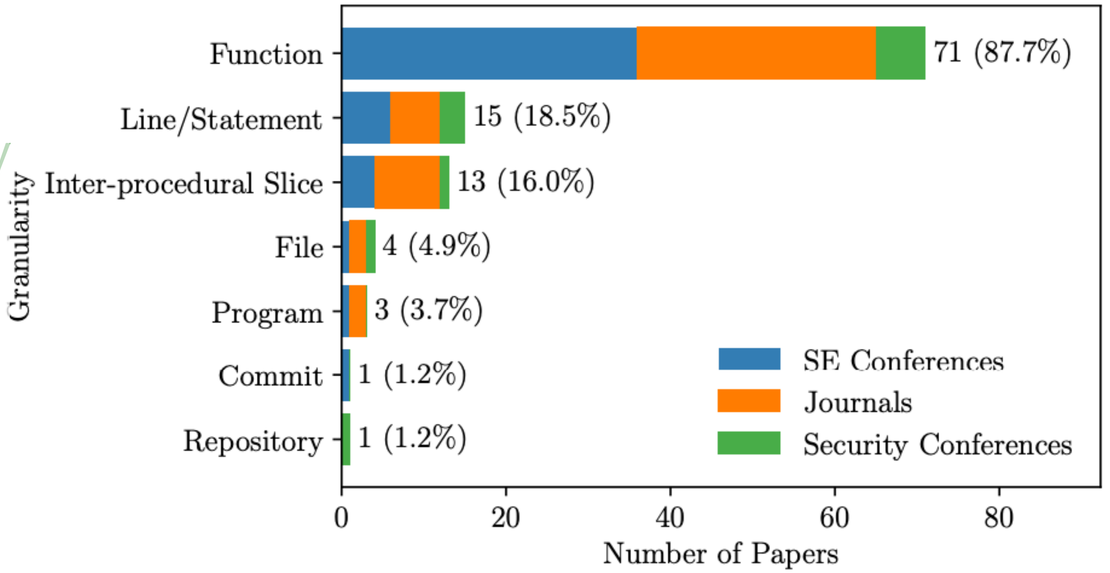
- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
    - Line/statement/commit-level: No reason to believe context-dependency problem is solved.
    - File/program-level: Impractical?



(a) ML4VD papers per problem statement granularity.

Top Score on the Wrong Exam

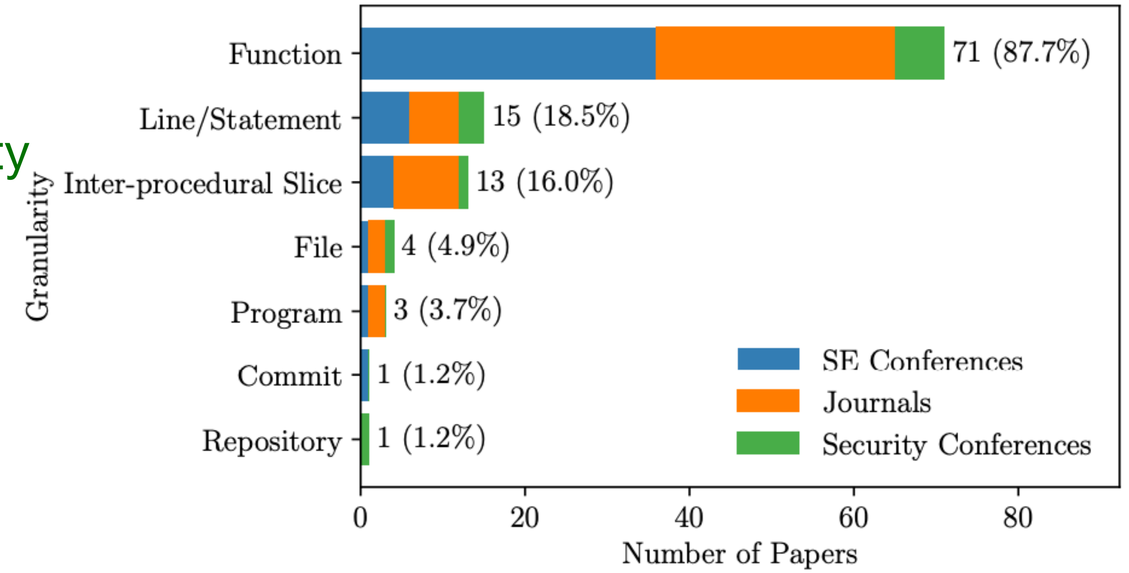
- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
    - Inter-procedural slice:
      - Solves context-dependency problem!
      - Delegates problem of deciding vulnerability to deciding slicing criterion.



(a) ML4VD papers per problem statement granularity.

Top Score on the Wrong Exam

- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
    - Inter-procedural slice:
      - Solves context-dependency problem!
      - Delegates problem of deciding vulnerability to deciding slicing criterion.



(a) ML4VD papers per problem statement granularity.

Top Score on the Wrong Exam

- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
  - Context-conditional classification.
    - Given the context of the program/repository, is this function vulnerable?
    - Problem:
      - Doesn’t solve our benchmarking problem (spurious correlations).
      - A bad classifier that *disregards* the context evidently still performs very well.

Top Score on the Wrong Exam

- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
  - Context-conditional classification.
    - Given the context of the program/repository, is this function vulnerable?
    - Problem:
      - Doesn’t solve our benchmarking problem (spurious correlations).
      - A bad classifier that *disregards* the context evidently still performs very well.

Top Score on the Wrong Exam

- What about alternative problem statements?
  - Classification with abstention.
  - Classification using other base units.
  - Context-conditional classification.
  - ML4VD as testing problem!

Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.

- Recommendation:
  - When benchmarking your technique, don’t blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.

- Recommendation:
  - When benchmarking your technique, don’t blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.

- Recommendation:
  - When benchmarking your technique, don't blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Top Score on the Wrong Exam

Niklas Risse
MPI-SP

ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany

According to our survey of machine learning for vulnerability detection (ML4VD), 9 in every 10 papers published in the past five years define ML4VD as a function-level binary classification problem:

Given a function, does it contain a security flaw?

From our experience as security researchers, faced with deciding whether a given function makes the program vulnerable to attacks, we would often first want to understand the context in which this function is called.

In this paper, we study how often this decision can really be made without further context and study both vulnerable and non-vulnerable functions in the most popular ML4VD datasets. We call a function "*vulnerable*" if it was involved in a patch of an actual security flaw and confirmed to cause the program's vulnerability. It is "*non-vulnerable*" otherwise. We find that in almost all cases this decision *cannot* be made without further context. Vulnerable functions are often vulnerable only *because* a corresponding vulnerability-inducing calling context exists while non-vulnerable functions would often be vulnerable *if* a corresponding context existed.

But why do ML4VD techniques achieve high scores even though there is demonstrably not enough information in these samples? Spurious correlations: We find that high scores can be achieved even when only word counts are available. This shows that these datasets can be exploited to achieve high scores without actually detecting any security vulnerabilities.

We conclude that the prevailing problem statement of ML4VD is ill-defined and call into question the internal validity of this growing body of work. Constructively, we call for more effective benchmarking methodologies to evaluate the true capabilities of ML4VD, propose alternative problem statements, and examine broader implications for the evaluation of machine learning and programming analysis research.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: machine learning, vulnerability detection, benchmark, function, LLM, data quality, context, spurious correlations, ML4VD, software security

ACM Reference Format:
Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA018 (July 2025), 23 pages. https://doi.org/10.1145/3728887

1 Introduction

In recent years, the number of papers published on the topic of machine learning for vulnerability detection (ML4VD) has dramatically increased. Because of this rise in popularity, the validity and soundness of the underlying methodologies and datasets becomes increasingly important. So then, how exactly is the problem of ML4VD defined and thus evaluated?

ACM 2994-970X/2025/7-ARTISSTA018
https://doi.org/10.1145/3728887

USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Recent results of machine learning for automatic vulnerability detection (ML4VD) have been very promising. Given only the source code of a function f, ML4VD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate ML4VD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating ML4VD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of ML4VD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six ML4VD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art ML4VD techniques are unable to distinguish vulnerable functions from their patches.

1 Introduction

Recently several different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1, 12–15, 28]. The resulting models seem to outperform traditional program analysis methods, e.g. static analysis, even without requiring any hard-coded knowledge of program semantics or computational models. So, does this mean that the problem of detecting security vulnerabilities in software is solved? Are these models actually able to detect security vulnerabilities, or do the reported scores provide a false sense of security?

Even though ML4VD techniques achieve high scores on vulnerability detection benchmark datasets, there are still situations in which they fail to meet expectations when presented with new data. For example, it is possible to apply small semantic preserving changes to augment the testing dataset of a state-of-the-art model and then measure whether the model changes its predictions. If it does, it would indicate a dependence of the prediction on unrelated features. Examples of such transformations are identifier renaming [18,38,39,41,42], insertion of unexecuted statements [18, 35, 39, 41] or replacement of code elements with equivalent elements [2, 21]. The impact of augmenting testing data using these transformations has been explored for many different software-related tasks and the results seem to be clear: Learning-based models fail to perform well when testing data gets augmented using semantic preserving transformations of code [2, 5, 18, 30, 35, 38, 39, 41, 42].

In our own experiments, we were able to reproduce the findings of the literature and made additional observations: ML4VD techniques that were trained on typical training data for vulnerability detection are also unable to distinguish between vulnerable functions and their patched counterparts. If a patched function is also predicted as vulnerable, this indicates that the prediction critically depends on features unrelated to the presence of a security vulnerability.

It has previously been proposed to reduce the dependence on unrelated features by augmenting not just the testing data but also the training data [5, 18, 35, 38, 39, 41, 42]. Indeed, this seems to restore the lost performance back to previous levels, but does it really reduce the dependence on unrelated features, or are the models just overfitting to different unrelated features of the data?

In this paper, we propose a novel benchmarking methodology that can be used to evaluate the capabilities of ML4VD techniques by using data augmentation. First, we propose

FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

ABSTRACT

Recent results of machine learning for automatic vulnerability detection have been very promising indeed: Given only the source code of a function f, models trained by machine learning techniques can decide if f contains a security flaw with up to 70% accuracy.

But how do we know that these results are general and not specific to the datasets? To study this question, researchers proposed to *amplify* the testing set by injecting semantic preserving changes and found that the model's accuracy significantly drops. In other words, the model uses *some* unrelated features during classification. In order to increase the robustness of the model, researchers proposed to train on amplified training data, and indeed model accuracy increased to previous levels.

In this paper, we replicate and continue this investigation, and provide an actionable model benchmarking methodology to help researchers better evaluate advances in machine learning for vulnerability detection. Specifically, we propose a cross validation algorithm, where a semantic preserving transformation is applied during the amplification of either the training set or the testing set. Using 11 transformations and 3 ML techniques, we find that the improved robustness only applies to the specific transformations used during training data amplification. In other words, the robustified models still rely on unrelated features for predicting the vulnerabilities in the testing data.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

machine learning, automatic vulnerability detection, semantic preserving transformations, large language models

ACM Reference Format:
Niklas Risse. 2023. Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3611643.3617845

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3617845

1 INTRODUCTION

Recently a number of different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1, 5–8, 14]. So, does this mean that the problem of detecting security vulnerabilities in software is solved? How do we know that the reported results are general and not specific to the benchmark datasets?

To study these questions, researchers have tried to explore the capabilities and limits of machine learning techniques in ways that go beyond simple evaluations on benchmark testing sets. For example, it is possible to apply small semantic preserving amplifications to the input programs of a state-of-the-art model and then measure, whether the model changes its predictions and whether it still performs well. Examples for such amplifications are identifier renaming [9, 17–20], insertion of unexecuted statements [9, 16, 18, 19] or replacement of code elements with equivalent elements [3, 10]. The impact of applying semantic preserving amplifications to testing data has been explored for many different tasks in software engineering, and the results seems to be clear: Machine learning techniques lack robustness against semantic preserving amplifications [3, 4, 9, 11, 15–20].

A common strategy to address the robustness problem is training data amplification; applying the same or similar amplifications to the training dataset. Many of the works that reported the lack of robustness of ML models when trained on unamplified data also investigated training data amplification using their respective methods [4, 9, 11, 16–20]. They found a restoration or at least improvement towards the initial high performance. But does training data amplification actually improve the ability of these models to detect vulnerabilities, or are they just overfitting to a different set of data?

We contribute to answering this question by proposing a general benchmarking methodology that can be used to evaluate the capabilities of machine learning models for vulnerability detection by using data amplification. The core of the methodology is a cross validation, in which a selected semantic preserving amplification method is applied to the training dataset of a model, and a *different* amplification method is applied to the testing dataset (see Figure 1). When repeated for all possible pairs out of a set of amplification methods, the resulting scores provide a measure of overfitting to the specific semantic preserving amplification methods that were used during training data amplification.

In addition to the general methodology, we present the results of an empirical study, in which we apply the proposed methodology to three state-of-the-art ML techniques for vulnerability detection. We implemented 11 different semantic preserving amplification methods and tried to cover types of amplifications commonly used

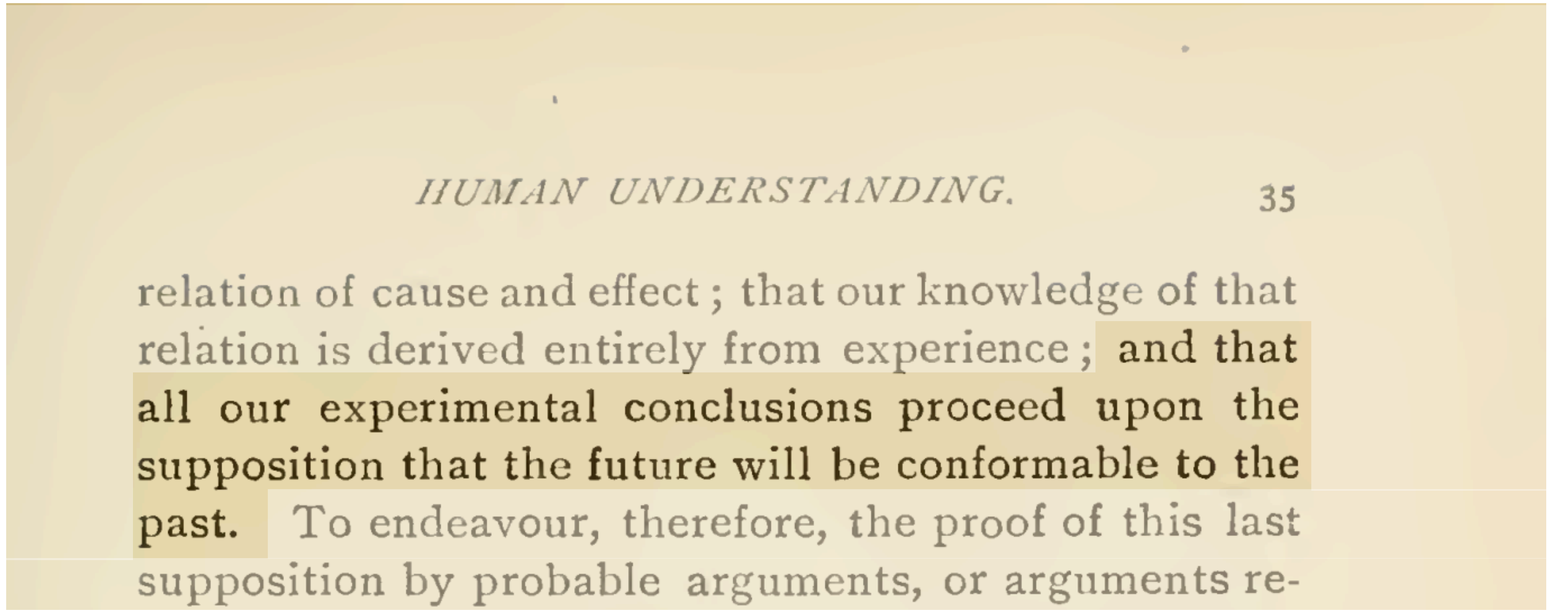
There are limits to benchmarking.

Section III

Philosophical Perspective

Hume's Problem of Induction

- We can never confirm a scientific theory just by collecting more evidence in favor.



HUMAN UNDERSTANDING. 35

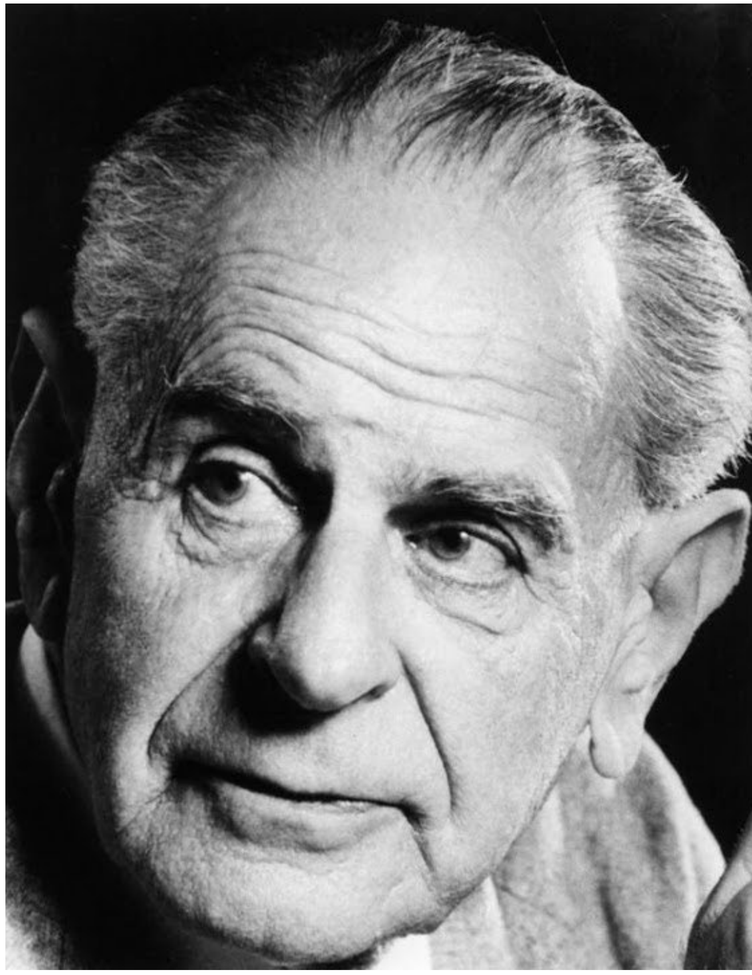
relation of cause and effect; that our knowledge of that relation is derived entirely from experience; and that all our experimental conclusions proceed upon the supposition that the future will be conformable to the past. To endeavour, therefore, the proof of this last supposition by probable arguments, or arguments re-

Enquiry Concerning Human Understanding (1748)

David Hume * 1711 in Edinburgh † 1776 in Edinburgh

Popper’s Critical Rationalism

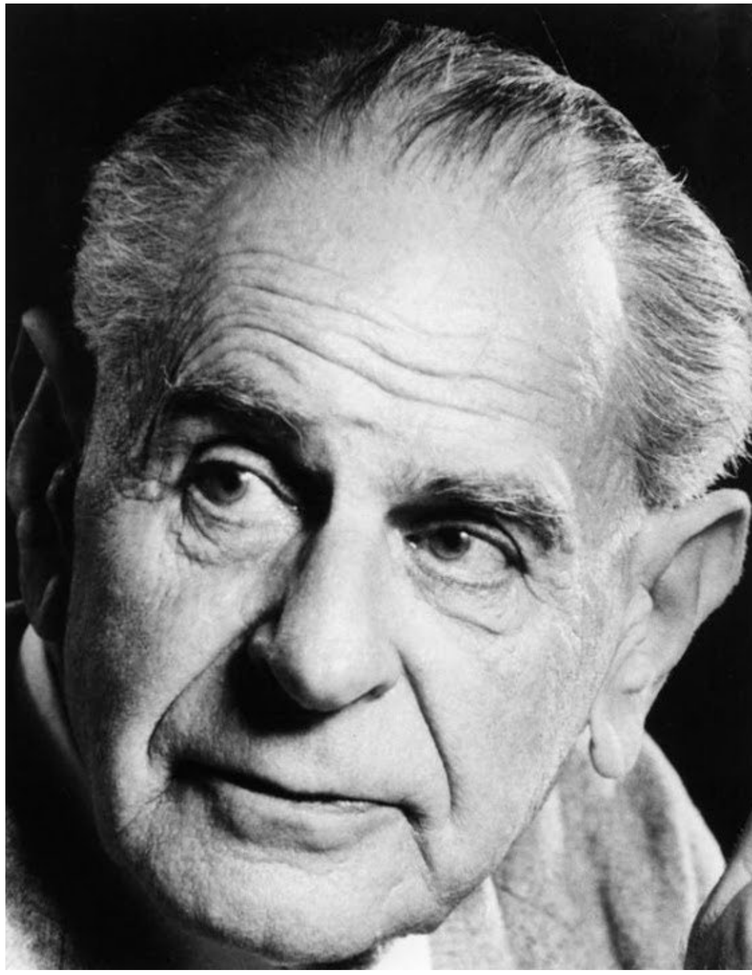
- We can never confirm a scientific theory just by collecting more evidence in favor.
- Popper’s criticial rationalism
  - Proposal for sound scientific progress in the absence of the possibility to confirm a scientific theory.
  - Instead of trying to confirm a theory, we should seriously attempt and fail to find counterexamples otherwise too many false theories remain in tact.



Karl Popper * 1902 in Vienna † 1994 in London

Popper’s Critical Rationalism

- We can never confirm a scientific theory just by collecting more evidence in favor.
- Popper’s criticial rationalism
  - Proposal for sound scientific progress in the absence of the possibility to confirm a scientific theory.
  - Instead of trying to confirm a theory, we should seriously attempt and fail to find counterexamples otherwise too many false theories remain in tact.



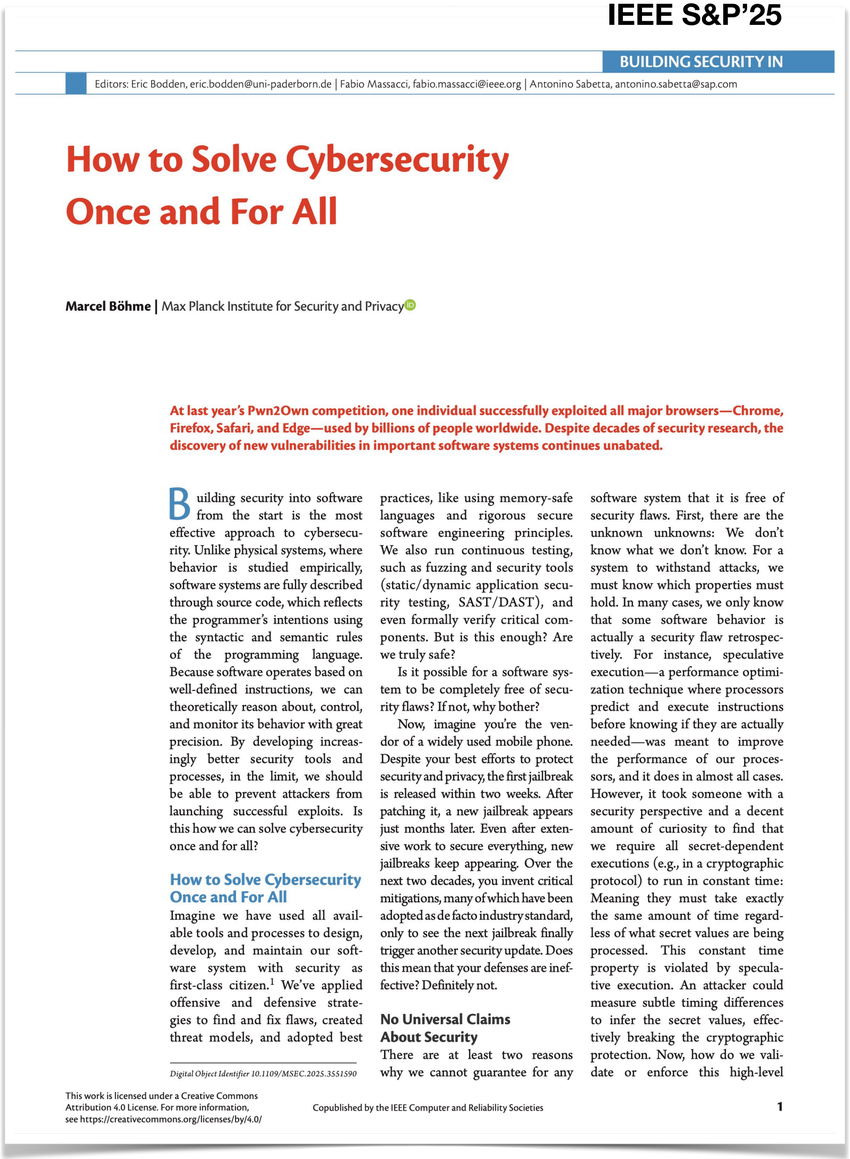
Karl Popper * 1902 in Vienna † 1994 in London

Benchmarking does not exempt us from Critical Rationalism.

- Benchmarking is us trying to confirm the progress of our techniques.
  - Benchmarking is important! Some empirical evidence is better than none!
- However, progress on a benchmark ≠ progress on the problem.
  - Going from 92% to 95% is no indicator of progress but of saturation.
  - Without an additional approach of critical rationalism, applied to both, our techniques as well as our benchmarking methodologies, too many ineffective techniques will appear to be effective.

Benchmarking does not exempt us from Critical Rationalism.

- Example: There is no guarantee of security.
  - Concretely, we can never hope to confirm the effectiveness of our defenses.
  - But we can seriously attempt and fail to find exploits in our software despite our defenses.



IEEE S&P'25

BUILDING SECURITY IN

Editors: Eric Bodden, eric.bodden@uni-paderborn.de | Fabio Massacci, fabio.massacci@ieee.org | Antonino Sabetta, antonino.sabetta@sap.com

How to Solve Cybersecurity Once and For All

Marcel Böhme | Max Planck Institute for Security and Privacy

At last year's Pwn2Own competition, one individual successfully exploited all major browsers—Chrome, Firefox, Safari, and Edge—used by billions of people worldwide. Despite decades of security research, the discovery of new vulnerabilities in important software systems continues unabated.

Building security into software from the start is the most effective approach to cybersecurity. Unlike physical systems, where behavior is studied empirically, software systems are fully described through source code, which reflects the programmer's intentions using the syntactic and semantic rules of the programming language. Because software operates based on well-defined instructions, we can theoretically reason about, control, and monitor its behavior with great precision. By developing increasingly better security tools and processes, in the limit, we should be able to prevent attackers from launching successful exploits. Is this how we can solve cybersecurity once and for all?

How to Solve Cybersecurity Once and For All

Imagine we have used all available tools and processes to design, develop, and maintain our software system with security as first-class citizen.[1] We've applied offensive and defensive strategies to find and fix flaws, created threat models, and adopted best practices, like using memory-safe languages and rigorous secure software engineering principles. We also run continuous testing, such as fuzzing and security tools (static/dynamic application security testing, SAST/DAST), and even formally verify critical components. But is this enough? Are we truly safe?

Is it possible for a software system to be completely free of security flaws? If not, why bother?

Now, imagine you're the vendor of a widely used mobile phone. Despite your best efforts to protect security and privacy, the first jailbreak is released within two weeks. After patching it, a new jailbreak appears just months later. Even after extensive work to secure everything, new jailbreaks keep appearing. Over the next two decades, you invent critical mitigations, many of which have been adopted as de facto industry standard, only to see the next jailbreak finally trigger another security update. Does this mean that your defenses are ineffective? Definitely not.

No Universal Claims About Security

There are at least two reasons why we cannot guarantee for any software system that it is free of security flaws. First, there are the unknown unknowns: We don't know what we don't know. For a system to withstand attacks, we must know which properties must hold. In many cases, we only know that some software behavior is actually a security flaw retrospectively. For instance, speculative execution—a performance optimization technique where processors predict and execute instructions before knowing if they are actually needed—was meant to improve the performance of our processors, and it does in almost all cases. However, it took someone with a security perspective and a decent amount of curiosity to find that we require all secret-dependent executions (e.g., in a cryptographic protocol) to run in constant time: Meaning they must take exactly the same amount of time regardless of what secret values are being processed. This constant time property is violated by speculative execution. An attacker could measure subtle timing differences to infer the secret values, effectively breaking the cryptographic protection. Now, how do we validate or enforce this high-level

Digital Object Identifier 10.1109/MSEC.2025.3551590

Copublished by the IEEE Computer and Reliability Societies

Measures are specific, our claims general.

ICSE'22

On the Reliability of Coverage-Based Fuzzer Benchmarking

Marcel Böhme
MPI-SP, Germany
Monash University, Australia

László Szekeres
Google, USA

Jonathan Metzman
Google, USA

ABSTRACT

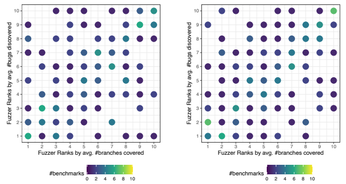
Given a program where none of our fuzzers finds any bugs, how do we know which fuzzer is better? In practice, we often look to code coverage as a proxy measure of fuzzer effectiveness and consider the fuzzer which achieves more coverage as the better one.

Indeed, evaluating 10 fuzzers for 23 hours on 24 programs, we find that a fuzzer that covers more code also finds more bugs. There is a *very strong correlation* between the coverage achieved and the number of bugs found by a fuzzer. Hence, it might seem reasonable to compare fuzzers in terms of coverage achieved, and from that derive empirical claims about a fuzzer's superiority at finding bugs.

Curiously enough, however, we find *no strong agreement* on which fuzzer is superior if we compared multiple fuzzers in terms of coverage achieved instead of the number of bugs found. The fuzzer best at achieving coverage, may not be best at finding bugs.

ACM Reference Format:
Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510230

1 INTRODUCTION



(a) 1 hour fuzzing campaigns ($\rho = 0.38$). **(b)** 1 day fuzzing campaigns ($\rho = 0.49$).

Figure 1: Scatterplot of the ranks of 10 fuzzers applied to 24 programs for (a) 1 hour and (b) 23 hours, when ranking each fuzzer in terms of the avg. number of branches covered (x-axis) and in terms of the avg. number of bugs found (y-axis).

Hence, it might seem reasonable to conjecture that the fuzzer which is better in terms of code coverage is also better in terms of bug finding—but is this really true? In Figure 1, we show the ranking of these fuzzers across all programs in terms of the average

Jonathan Metzman
Google

László Szekeres
Google

Benchmarks are specific, our claims general.

TOSEM'25

Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome.* This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization,* can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

Dylan Wolff
NUS

Abhik Roychoudhury
NUS

Benchmarking confirms effectiveness. What about its limits?

ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany

According to our survey of machine learning for vulnerability detection (MLVD), 9 in every 10 papers published in the past five years define MLVD as a function-level binary classification problem:

Given a function, does it contain a security flaw?

From our experience as security researchers, faced with deciding whether a given function makes the program vulnerable to attacks, we would often first want to understand the context in which this function is called.

In this paper, we study how often this decision can really be made without further context and study both vulnerable and non-vulnerable functions in the most popular MLVD datasets. We call a function "vulnerable" if it was involved in a patch of an actual security flaw and confirmed to cause the program's vulnerability. It is "non-vulnerable" otherwise. We find that in almost all cases this decision cannot be made without further context. Vulnerable functions are often vulnerable only *because* a corresponding vulnerability-inducing calling context exists while non-vulnerable functions would often be vulnerable if a corresponding context existed.

But why do MLVD techniques achieve high scores even though there is demonstrably not enough information in these samples? Spurious correlations: We find that high scores can be achieved even when only word counts are available. This shows that these datasets can be exploited to achieve high scores without actually detecting any security vulnerabilities.

We conclude that the prevailing problem statement of MLVD is ill-defined and call into question the internal validity of this growing body of work. Constructively, we call for more effective benchmarking methodologies to evaluate the true capabilities of MLVD, propose alternative problem statements, and examine broader implications for the evaluation of machine learning and programming analysis research.

CCS Concepts: • **Security and privacy** → **Software and application security**; • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: machine learning, vulnerability detection, benchmark, function, LLM, data quality, context, spurious correlations, MLVD, software security

ACM Reference Format:
Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA018 (July 2025), 23 pages. https://doi.org/10.1145/3728887

1 Introduction

In recent years, the number of papers published on the topic of machine learning for vulnerability detection (MLVD) has dramatically increased. Because of this rise in popularity, the validity and soundness of the underlying methodologies and datasets becomes increasingly important. So then, how exactly is the problem of MLVD defined and thus evaluated?

ACM 2994-970X/2025/7-ARTISSTA018
https://doi.org/10.1145/3728887

USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Recent results of machine learning for automatic vulnerability detection (MLVD) have been very promising. Given only the source code of a function f, MLVD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate MLVD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating MLVD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of MLVD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six MLVD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art MLVD techniques are unable to distinguish vulnerable functions from their patches.

1 Introduction

Recently several different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1,12–15,28]. The resulting models seem to outperform traditional program analysis methods, e.g. static analysis, even without requiring any hard-coded knowledge of program semantics or computational models. So, does this mean that the problem of detecting security vulnerabilities in software is solved? Are these models actually able to detect security vulnerabilities, or do the reported scores provide a false sense of security?

Even though ML-VD techniques achieve high scores on vulnerability detection benchmark datasets, there are still situations in which they fail to meet expectations when presented with new data. For example, it is possible to apply small semantic preserving changes to augment the testing dataset of a state-of-the-art model and then measure whether the model changes its predictions. If it does, it would indicate a dependence of the prediction on unrelated features. Examples of such transformations are identifier renaming [18,38,39,41,42], insertion of unexecuted statements [18, 35, 39, 41] or replacement of code elements with equivalent elements [2,21]. The impact of augmenting testing data using these transformations has been explored for many different software-related tasks and the results seem to be clear: Learning-based models fail to perform well when testing data gets augmented using semantic preserving transformations of code [2, 5, 18, 30, 35, 38, 39, 41, 42].

In our own experiments, we were able to reproduce the findings of the literature and made additional observations: ML-VD techniques that were trained on typical training data for vulnerability detection are also unable to distinguish between vulnerable functions and their patched counterparts. If a patched function is also predicted as vulnerable, this indicates that the prediction critically depends on features unrelated to the presence of a security vulnerability.

It has previously been proposed to reduce the dependence on unrelated features by augmenting not just the testing data but also the training data [5, 18, 35, 38, 39, 41, 42]. Indeed, this seems to restore the test performance back to previous levels, but does it really reduce the dependence on unrelated features, or are the models just overfitting to different unrelated features of the data?

In this paper, we propose a novel benchmarking methodology that can be used to evaluate the capabilities of MLVD techniques by using data augmentation. First, we propose

Niklas Risse
MPI-SP

FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

ABSTRACT

Recent results of machine learning for automatic vulnerability detection have been very promising. Given only the source code of a function f, ML-VD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate ML-VD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating ML-VD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of ML-VD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six ML-VD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art ML-VD techniques are unable to distinguish vulnerable functions from their patches.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

machine learning, automatic vulnerability detection, semantic preserving transformations, large language models

ACM Reference Format:
Niklas Risse. 2023. Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3611643.3613892

ACM ISBN 979-8-4007-0327-0/23/12
https://doi.org/10.1145/3611643.3613892

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.
- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

TOSEM'25

Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome*. This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization*, can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

Dylan Wolff
NUS

Abhik Roychoudhury
NUS

Benchmarking confirms effectiveness. What about its limits?

Niklas Risse
MPI-SP

ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany

USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse, MPI-SP, Germany
Marcel Böhme, MPI-SP, Germany

FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Unless there is high agreement between two measures, report multiple measures of success so as to triangulate.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

TOSEM'25

Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties

DYLAN WOLFF, National University of Singapore, Singapore
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

In a typical experimental design in fuzzing, we would run two or more fuzzers on an appropriate set of benchmark programs plus seed corpora and consider their ranking in terms of code coverage or bugs found as outcome. However, the specific characteristics of the benchmark setup clearly can have some impact on the benchmark outcome. If the programs were larger, or these initial seeds were chosen differently, the same fuzzers may be ranked differently; the benchmark outcome would change. In this paper, we explore two methodologies to *quantify the impact of the specific properties on the benchmarking outcome.* This allows us to report the benchmarking outcome counter-factually, e.g., "If the benchmark had larger programs, this fuzzer would outperform all others". Our first methodology is the *controlled experiment* to identify a causal relationship between a single property in isolation and the benchmarking outcome. The controlled experiment requires manually altering the fuzzer or system under test to vary that property while holding all other variables constant. By repeating this controlled experiment for multiple fuzzer implementations, we can gain detailed insights to the different effects this property has on various fuzzers. However, due to the large number of properties and the difficulty of realistically manipulating one property exactly, control may not always be practical or possible. Hence, our second methodology is *randomization* and non-parametric regression to identify the strength of the relationship between arbitrary benchmark properties (i.e., covariates) and outcome. Together, these two fundamental aspects of experimental design, *control* and *randomization*, can provide a comprehensive picture of the impact of various properties of the current benchmark on the fuzzer ranking. These analyses can be used to guide fuzzer developers towards areas of improvement in their tools and allow researchers to make more nuanced claims about fuzzer effectiveness. We instantiate each approach on a subset of properties suspected of impacting the relative effectiveness of fuzzers and quantify the effects of these properties on the evaluation outcome. In doing so, we identify multiple properties, such as the coverage of the initial seed-corpus and the program execution speed, which can have statistically significant effect on the *relative* effectiveness of fuzzers.

Dylan Wolff
NUS

Abhik Roychoudhury
NUS

Benchmarking confirms effectiveness. What about its limits?

Niklas Risse
MPI-SP

ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany

According to our survey of machine learning for vulnerability detection (MLVD), 9 in every 10 papers published in the past five years define MLVD as a function-level binary classification problem:

Given a function, does it contain a security flaw?

From our experience as security researchers, faced with deciding whether a given function makes the program vulnerable to attacks, we would often first want to understand the context in which this function is called.

In this paper, we study how often this decision can really be made without further context and study both vulnerable and non-vulnerable functions in the most popular MLVD datasets. We call a function "vulnerable" if it was involved in a patch of an actual security flaw and confirmed to cause the program's vulnerability. It is "non-vulnerable" otherwise. We find that in almost all cases this decision *cannot* be made without further context. Vulnerable functions are often vulnerable only *because* a corresponding vulnerability-inducing calling context exists while non-vulnerable functions would often be vulnerable *if* a corresponding context existed.

But why do MLVD techniques achieve high scores even though there is demonstrably not enough information in these samples? Spurious correlations: We find that high scores can be achieved even when only word counts are available. This shows that these datasets can be exploited to achieve high scores without actually detecting any security vulnerabilities.

We conclude that the prevailing problem statement of MLVD is ill-defined and call into question the internal validity of this growing body of work. Constructively, we call for more effective benchmarking methodologies to evaluate the true capabilities of MLVD, propose alternative problem statements, and examine broader implications for the evaluation of machine learning and programming analysis research.

CCS Concepts: • **Security and privacy → Software and application security**; • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: machine learning, vulnerability detection, benchmark, function, LLM, data quality, context, spurious correlations, MLVD, software security

ACM Reference Format:
Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA018 (July 2025), 25 pages. https://doi.org/10.1145/3728887

1 Introduction

In recent years, the number of papers published on the topic of machine learning for vulnerability detection (MLVD) has dramatically increased. Because of this rise in popularity, the validity and soundness of the underlying methodologies and datasets becomes increasingly important. So then, how exactly is the problem of MLVD defined and thus evaluated?

ACM 2994-970X/2025/7-ARTISSTA018
https://doi.org/10.1145/3728887

USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Recent results of machine learning for automatic vulnerability detection (MLVD) have been very promising. Given only the source code of a function f, MLVD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate MLVD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating MLVD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of MLVD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six MLVD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art MLVD techniques are unable to distinguish vulnerable functions from their patches.

1 Introduction

Recently several different publications have reported high scores on vulnerability detection benchmarks using machine learning (ML) techniques [1,12–15,28]. The resulting models seem to outperform traditional program analysis methods, e.g., static analysis, even without requiring any hard-coded knowledge of program semantics or computational models. So, does this mean that the problem of detecting security vulnerabilities in software is solved? Are these models actually able to detect security vulnerabilities, or do the reported scores provide a false sense of security?

Even though MLVD techniques achieve high scores on vulnerability detection benchmark datasets, there are still situations in which they fail to meet expectations when presented with new data. For example, it is possible to apply small semantic preserving changes to augment the testing dataset of a state-of-the-art model and then measure whether the model changes its predictions. If it does, it would indicate a dependence of the prediction on unrelated features. Examples of such transformations are identifier renaming [18,38,39,41,42], insertion of unexecuted statements [18, 35, 39, 41] or replacement of code elements with equivalent elements [2,21]. The impact of augmenting testing data using these transformations has been explored for many different software-related tasks and the results seem to be clear: Learning-based models fail to perform well when testing data gets augmented using semantic preserving transformations of code [2, 5, 18, 30, 35, 38, 39, 41, 42].

In our own experiments, we were able to reproduce the findings of the literature and made additional observations: MLVD techniques that were trained on typical training data for vulnerability detection are also unable to distinguish between vulnerable functions and their patched counterparts. If a patched function is also predicted as vulnerable, this indicates that the prediction critically depends on features unrelated to the presence of a security vulnerability.

It has previously been proposed to reduce the dependence on unrelated features by augmenting not just the testing data but also the training data [5, 18, 35, 38, 39, 41, 42]. Indeed, this seems to restore the lost performance back to previous levels, but does it really reduce the dependence on unrelated features, or are the models just overfitting to different unrelated features of the data?

In this paper, we propose a novel benchmarking methodology that can be used to evaluate the capabilities of MLVD techniques by using data augmentation. First, we propose

FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

ABSTRACT

Recent results of machine learning for automatic vulnerability detection have been very promising. Given only the source code of a function f, MLVD techniques can decide if f contains a security flaw with up to 70% accuracy. However, as evident in our own experiments, the same top-performing models are unable to distinguish between functions that contain a vulnerability and functions where the vulnerability is patched. So, how can we explain this contradiction and how can we improve the way we evaluate MLVD techniques to get a better picture of their actual capabilities?

In this paper, we identify overfitting to unrelated features and out-of-distribution generalization as two problems, which are not captured by the traditional approach of evaluating MLVD techniques. As a remedy, we propose a novel benchmarking methodology to help researchers better evaluate the true capabilities and limits of MLVD techniques. Specifically, we propose (i) to augment the training and validation dataset according to our cross-validation algorithm, where a semantic preserving transformation is applied during the augmentation of either the training set or the testing set, and (ii) to augment the testing set with code snippets where the vulnerabilities are patched.

Using six MLVD techniques and two datasets, we find (a) that state-of-the-art models severely overfit to unrelated features for predicting the vulnerabilities in the testing data, (b) that the performance gained by data augmentation does not generalize beyond the specific augmentations applied during training, and (c) that state-of-the-art MLVD techniques are unable to distinguish vulnerable functions from their patches.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software testing and debugging**

KEYWORDS

machine learning, automatic vulnerability detection, semantic preserving transformations, large language models

ACM Reference Format:
Niklas Risse. 2023. Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA, ACM, New York, NY, USA,* 3 pages. https://doi.org/10.1145/3611643.3613871

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM 979-8-4007-0327-0/23/12
https://doi.org/10.1145/3611643.3613871

Conduct counterfactual analysis.
Report conditions under which benchmark outcome changes.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

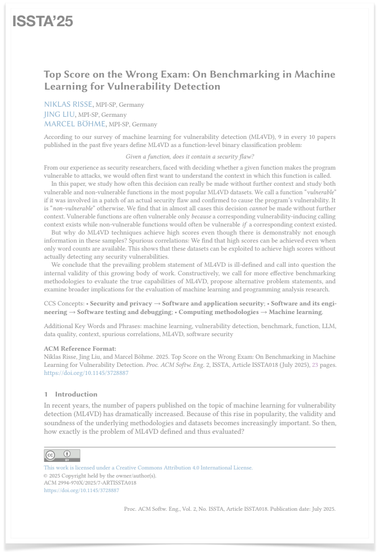
- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

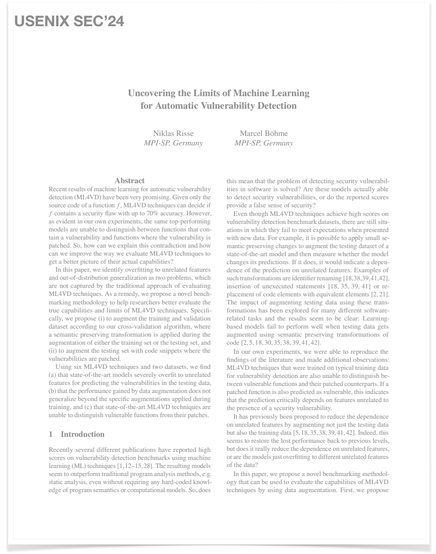
- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarking confirms effectiveness. What about its limits?


ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany


USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse, MPI-SP, Germany
Marcel Böhme, MPI-SP, Germany

Niklas Risse
MPI-SP


FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

Conduct counterfactual analysis.
Report conditions under which
benchmark outcome changes.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

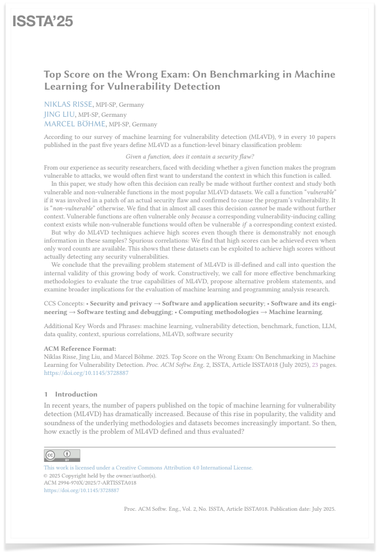
- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

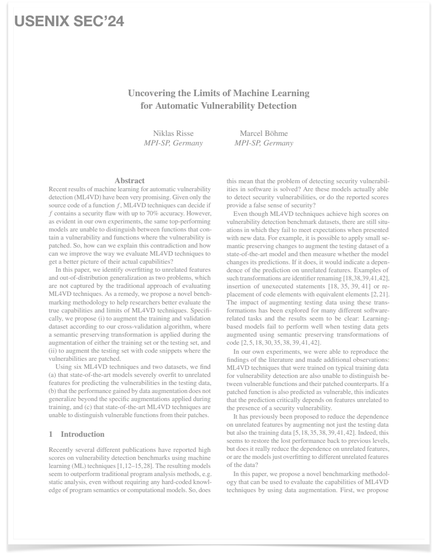
- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Benchmarking confirms effectiveness. What about its limits?


ISSTA'25

Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection

NIKLAS RISSE, MPI-SP, Germany
JING LIU, MPI-SP, Germany
MARCEL BÖHME, MPI-SP, Germany


USENIX SEC'24

Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection

Niklas Risse, MPI-SP, Germany
Marcel Böhme, MPI-SP, Germany

Niklas Risse
MPI-SP


FSE'23 Student Research Competition

Detecting Overfitting of Machine Learning Techniques for Automatic Vulnerability Detection

Niklas Risse
niklas.risse@mpi-sp.org
Max-Planck-Institute for Security and Privacy
Bochum, Germany

Step back and reflect if we are asking the right questions to begin with.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.
- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.
- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.
- Recommendation:
  - When benchmarking your technique, don't blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Step back and reflect if we are asking the right questions to begin with.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.
- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.
- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.
- Recommendation:
  - When benchmarking your technique, don't blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Benchmarks induce progress.

- Benchmarking to measure progress in all of automation.
  - Automated Software Engineering: SWE-Bench, Defects4J, CoREBench.
  - Automated Cybersecurity: DARPA CGC, AIxCC (8.5 *million* USD in prizes)
  - Machine Learning / Artificial Intelligence:
    - ARC Challenge (1+ million USD in prizes).
    - Most ML/AI conferences have a track to announce new benchmarks.
    - Every announcement of a new LLM comes with results on popular benchmarks.

Measures are specific, our claims general.

- What did we learn?
  - Sometimes, there is no optimal measure of success.
  - Even if there is a strong correlation, you cannot substitute one measure for another and expect the same benchmarking outcome.

- Recommendation:
  - Triangulate effectiveness using different measures of success.
  - Unless there is agreement between two measures, report both measures.

Benchmarks are specific, our claims general.

- What did we learn?
  - Your benchmarking outcome is specific to your benchmark configuration.
  - Techniques might seem to perform similar on the average instance. Atomistic benchmarking hides the strengths of individual techniques.

- Recommendation:
  - Conduct a counterfactual analysis to report the conditions under which a benchmark outcome changes.

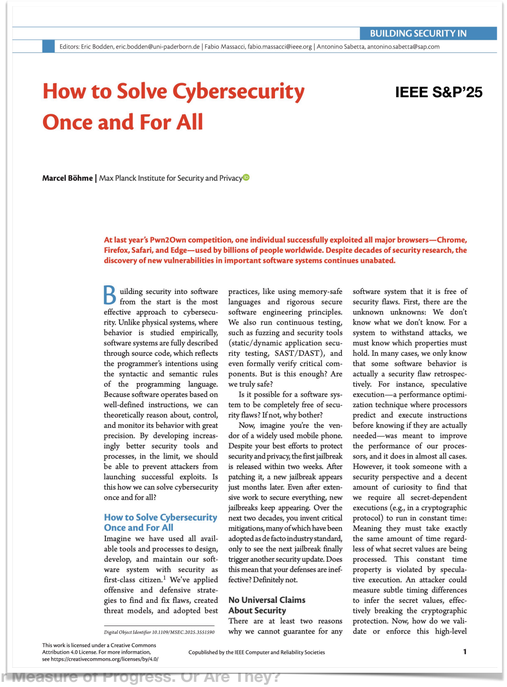
Top Score on the Wrong Exam

- What did we learn?
  - We use benchmarking to learn how well a technique solves **the problem**, **but** an entire field can beat benchmarks without solving **the problem.**
  - For ML techniques, we **must** tackle the problem of spurious correlations before we can consider benchmark outcomes as trustworthy.

- Recommendation:
  - When benchmarking your technique, don't blindly trust the numbers. Step back and reflect if you are asking the right questions to begin with.

Benchmarking does not exempt us from Critical Rationalism.

- We should stop only trying to confirm the effectiveness of our techniques and start failing to find important counterexamples.



BUILDING SECURITY IN

Editors: Eric Bodden, eric.bodden@uni-paderborn.de | Fabio Massacci, fabio.massacci@ieee.org | Antonino Sabetta, antonino.sabetta@sap.com

How to Solve Cybersecurity Once and For All

IEEE S&P'25

Marcel Böhme | Max Planck Institute for Security and Privacy

At last year's Pwn2Own competition, one individual successfully exploited all major browsers—Chrome, Firefox, Safari, and Edge—used by billions of people worldwide. Despite decades of security research, the discovery of new vulnerabilities in important software systems continues unabated.

Building security into software from the start is the most effective approach to cybersecurity. Unlike physical systems, where behavior is studied empirically, software systems are fully described through source code, which reflects the programmer's intentions using the syntactic and semantic rules of the programming language. Because software operates based on well-defined instructions, we can theoretically reason about, control, and monitor its behavior with great precision. By developing increasingly better security tools and processes, in the limit, we should be able to prevent attackers from launching successful exploits. Is this how we can solve cybersecurity once and for all?

How to Solve Cybersecurity Once and For All

Imagine we have used all available tools and processes to design, develop, and maintain our software system with security as first-class citizen.[1] We've applied offensive and defensive strategies to find and fix flaws, created threat models, and adopted best practices, like using memory-safe languages and rigorous secure software engineering principles. We also run continuous testing, such as fuzzing and security tools (static/dynamic application security testing, SAST/DAST), and even formally verify critical components. But is this enough? Are we truly safe?

Is it possible for a software system to be completely free of security flaws? If not, why bother?

Now, imagine you're the vendor of a widely used mobile phone. Despite your best efforts to protect security and privacy, the first jailbreak is released within two weeks. After patching it, a new jailbreak appears just months later. Even after extensive work to secure everything, new jailbreaks keep appearing. Over the next two decades, you invest critical mitigations, many of which have been adopted as de facto industry standard, only to see the next jailbreak finally trigger another security update. Does this mean that your defenses are ineffective? Definitely not.

No Universal Claims About Security

There are at least two reasons why we cannot guarantee for any software system that it is free of security flaws. First, there are the unknown unknowns: We don't know what we don't know. For a system to withstand attacks, we must know which properties must hold. In many cases, we only know that some software behavior is actually a security flaw retrospectively. For instance, speculative execution—a performance optimization technique where processors predict and execute instructions before knowing if they are actually needed—was meant to improve the performance of our processors, and it does in almost all cases. However, it took someone with a security perspective and a decent amount of curiosity to find that we require all secret-dependent executions (e.g., in a cryptographic protocol) to run in constant time: Meaning they must take exactly the same amount of time regardless of what secret values are being processed. This constant time property is violated by speculative execution. An attacker could measure subtle timing differences to infer the secret values, effectively breaking the cryptographic protection. Now, how do we validate or enforce this high-level

Digital Object Identifier 10.1109/MSEC.2025.3533390

Copublished by the IEEE Computer and Reliability Societies