School of Computation, Information and Technology

Technical University of Munich

Dissertation in Informatics

Refining Code Coverage to Better Reflect Fuzzing Performance

Stephan Lipp

Technische Universität München
TUM School of Computation, Information and Technology

Refining Code Coverage to Better Reflect Fuzzing Performance

Stephan Lipp

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Prof. Dr. Alexander Pretschner
2. Prof. Dr. Cristian Cadar
3. Prof. Dr. Gordon Fraser

Die Dissertation wurde am 28.03.2025 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 30.10.2025 angenommen.

Acknowledgments

First and foremost, I would like to thank my supervisor, Alexander Pretschner, for giving me the opportunity to pursue a Ph.D. in software testing in his group. His guidance has profoundly shaped my approach to research, emphasizing both scientific rigor and practical relevance. I am particularly grateful for his advice to take a step back and look at the *bigger picture* of a complex problem to not get lost in the details.

I am also very grateful to Cristian Cadar and Gordon Fraser, who kindly took on the role of additional supervisors. I am especially thankful to Cristian for his insightful feedback during my visit to his research group, which was very helpful and had a lasting impact on my work.

A heartfelt thank you goes to my Ph.D. mentor, Sebastian, for his continuous support and motivation over the past few years. The many discussions we had were crucial in helping me overcome research obstacles that, at times, seemed insurmountable.

I would also like to thank Marcel Böhme, with whom I have had the pleasure of collaborating on several very interesting projects. Throughout these projects, I have always been inspired by his creativity in approaching challenging research problems. His support and constructive feedback have thereby also greatly improved my own research.

My time spent at the chair would not have been the same without the many wonderful colleagues from whom I have learned so much. I am fortunate to say that many of you have become dear friends over the years, which I consider one of the greatest rewards of this journey. A special thanks to my former office mate and good friend, Severin, with whom I shared an office for most of my time at the chair, which made this experience all the more enjoyable. *Thank you, Severin!*

Also, a big thank you to Kohei, David, and Alex, as well as Severin, Marcel, and Sebastian, for proofreading earlier versions of this dissertation, which greatly contributed to its quality.

Last but certainly not least, I would like to express my deepest gratitude to my family for their unwavering support and encouragement, which has been the foundation of this achievement. Completing this Ph.D. would not have been possible without you.

Zusammenfassung

Kontext. Fuzzing ist eine Sicherheitstesttechnik, die durch einen Fuzzer automatisiert wird und während einer Fuzzing-Kampagne das zu testende System wiederholt mit zufälligen, fehlerhaften Testeingaben ausführt. Ziel ist es, Sicherheitslücken—insbesondere in C/C++ Software—aufzudecken, die bei einer Ausnutzung erheblichen Schaden verursachen können. Ein wichtiger Bereich der Fuzzing-Forschung konzentriert sich dabei auf Metriken, welche die Fähigkeit einer Kampagne bewerten, alle Sicherheitsschwachstellen im System aufzudecken. Dies ist nicht nur relevant, um zu bestimmen, wann eine einzelne Kampagne oder Fuzzing im Allgemeinen beendet werden soll, sondern auch, um die Effektivität der vom Fuzzer generierten Tests zu optimieren.

Problemdomäne. Die Codeabdeckung ist die am weitesten verbreitete Metrik im Softwaretesten und misst den Anteil des Quellcodes des zu testenden Systems, der durch die Tests mindestens einmal ausgeführt wurde. Die Seltenheit von Sicherheitslücken und die Gleichbehandlung aller Codebereiche durch die Codeabdeckung führen jedoch häufig zu einer ungenauen Bewertung der Kampagnenleistung. Dies kann zu übermäßig langen und kostspieligen Kampagnen führen oder—schlimmer noch—zu vorzeitig abgebrochenen Kampagnen, die kritische Schwachstellen übersehen. Ziel dieser Abschlussarbeit ist es daher, eine neue Metrik zu entwickeln, welche die Leistung des Fuzzings besser widerspiegelt als die Codeabdeckung.

Forschungslücken. Um dieses Problem anzugehen, schlagen wir vor, die Codeabdeckung dahingehend zu verfeinern, so dass sie sich nur auf den potentiell (sicherheits-)anfälligen Code konzentriert. Die hier identifizierten Forschungslücken beziehen sich dabei auf (statische) Schwachstellenvorhersage und Fuzzing. Bezüglich Ersterer fehlt es an Studien, die Static Application Security Testing (SAST)-Techniken zur Erkennung von anfälligem Code unter realen Bedingungen untersuchen. Bezüglich Letzterer gibt es bisher keine Arbeiten, die eine präzise Schwachstellenvorhersage nutzen, um eine schwachstellenorientierte Codeabdeckungsmetrik zu erstellen, die eine bessere Fuzzing-Bewertung und damit eine effizientere und effektivere Kampagnenbeendigung und -steuerung ermöglicht.

Lösungsansatz. Um die identifizierten Lücken zu schließen, evaluieren wir zunächst die Effektivität moderner C/C++ SAST-Tools, um deren Stärken und Schwächen zu verstehen. Anschließend beheben wir deren Unzulänglichkeiten, indem wir SAST mit Codekomplexitätsmetriken mittels maschinellem Lernen (ML) kombinieren, was uns erlaubt, anfälligen Code für unsere vorgeschlagene Metrik zuverlässig zu erkennen. Wir verwenden dann unsere Metrik, um Kampagnen dynamisch zu beenden, wenn ihre Effektivität über einen längeren Zeitraum stagniert, wobei wir einen neuartigen Coverage-Analyzer einsetzen, um die Kampagne in Echtzeit zu überwachen. Schließlich integrieren wir unsere Metrik in einen Fuzzer, um die Testgenerierung zu optimieren.

Beiträge. Zusammenfassend leisten wir einen Forschungsbeitrag in Form einer detaillierten Evaluierung von SAST-Tools sowie eines ML-basierten Ansatzes zur Schwach-

stellenvorhersage, welcher die SAST-Schwächen adressiert und die Erstellung unserer Fuzzing-Metrik ermöglicht. Darüber hinaus tragen wir ein schwachstellenorientiertes Abbruch- und Testoptimierungskriterium bei, das im Vergleich zur regulären Codeabdeckung ein früheres Beenden von Kampagnen (bei geringem Risiko, Schwachstellen zu übersehen) bzw. eine effektivere Aufdeckung von Sicherheitslücken ermöglicht. Diese Ergebnisse zeigen somit die verbesserte Genauigkeit unserer Metrik bei der Bewertung der (Kosten-)Effizienz und Effektivität von Fuzzing, was den Meta-Beitrag dieser Dissertation darstellt.

Abstract

Context. Fuzzing is a security testing technique, automated by a fuzzer, that repeatedly executes the system under test (SUT) during a fuzzing campaign with randomized, malformed test inputs. Its goal is to uncover security vulnerabilities, especially in C/C++ software, that can cause severe damage if exploited. An important area of fuzzing research focuses on metrics that assess a campaign's ability to detect all the security bugs in the SUT. This is relevant not only for determining when to stop fuzzing but also for optimizing the effectiveness of the fuzzer-generated tests.

Problem Domain. Code coverage is the most widely used metric in software testing, measuring the proportion of the SUT's source code executed at least once by the tests. However, the scarcity of security bugs, coupled with code coverage treating all code regions as equally important, often leads to an inaccurate assessment of the campaign's performance. This can result in overly long and costly campaigns or, worse, prematurely terminated campaigns that miss critical vulnerabilities. Therefore, in this thesis, we aim to develop a novel metric that better reflects the performance of fuzzing than code coverage.

Research Gaps. To address this problem, we propose refining code coverage by focusing only on the potentially vulnerable code. The identified research gaps thereby relate to (static) vulnerability prediction and fuzzing. Regarding the former, there is a lack of studies that examine static application security testing (SAST) techniques for detecting bug-prone code under real-world conditions. Regarding the latter, no prior work has used precise vulnerability prediction to create a vulnerability-oriented code coverage metric that provides better fuzzing assessments and, hence, more efficient and effective campaign termination and guidance.

Solution. To address the identified gaps, we first evaluate the effectiveness of modern C/C++ SAST tools to understand their strengths and weaknesses. Next, we resolve their shortcomings by combining SAST with code complexity metrics using machine learning (ML), allowing us to reliably detect bug-prone code for our proposed metric. We then use our metric to dynamically stop campaigns when their effectiveness stagnates for an extended period, thereby using a new coverage analyzer to monitor the campaign in real-time. Finally, we integrate our metric into a fuzzer to optimize the test generation.

Contributions. Overall, we contribute an in-depth SAST tool evaluation along with an ML-based vulnerability prediction approach that addresses their deficiencies and enables the creation of our fuzzing metric. We also contribute a vulnerability-oriented stopping and test-optimization criterion that, compared to regular code coverage, allows campaigns to be terminated earlier (with a low risk of missing bugs) and detects security bugs more effectively, respectively. Thus, these results show our metric's improved accuracy in assessing the (cost-)efficiency and effectiveness of fuzzing, which is the meta-contribution of this dissertation.

Outline of the Thesis

Chapter 1 – Introduction

This chapter motivates the topic of this dissertation, outlines the research problem and gaps identified in the literature, and presents the solutions and contributions to fill them. Parts of this chapter have appeared in peer-reviewed publications [168–171], first-authored by the author of this thesis.

Chapter 2 – Background

This chapter provides the background knowledge required to understand the subsequent chapters. It covers the basic concepts, methodological and technical details, and quality criteria for static and predictive program analysis, as well as fuzzing. Parts of this chapter have appeared in peer-reviewed publications [168, 170, 171], first-authored by the author of this thesis.

Chapter 3 – SAST Tool Evaluation

This chapter empirically evaluates the effectiveness of several advanced static application security testing (SAST) tools in identifying potentially vulnerable code in real-world subject programs with known vulnerabilities. Parts of this chapter have appeared in a peer-reviewed publication [168], first-authored by the author of this thesis.

Chapter 4 – ML-Based Vulnerability Prediction

This chapter researches the integration of code complexity metrics and SAST tool findings into a vulnerability prediction model using machine learning to identify potentially vulnerable functions more reliably and accurately than using SAST tools alone. Parts of this chapter have appeared in a peer-reviewed publication [170], first-authored by the author of this thesis.

Chapter 5 – Real-Time Coverage Tracking

This chapter presents the implementation, application, and practical benefits of our fuzzer-independent coverage analyzer, which allows real-time monitoring and, thus, dynamic termination of fuzzing campaigns. It also describes a comprehensive coverage dataset created with our tool that can be used for further empirical fuzzing research. Parts of this chapter have appeared in a peer-reviewed publication [169], first-authored by the author of this thesis.

Chapter 6 – Vulnerability-Oriented Fuzzing Stopping Criterion

This chapter presents a novel stopping criterion that terminates fuzzing campaigns based on the saturation of our vulnerability-oriented code coverage metric. By analyzing the tradeoff between fuzzing time saved and security bugs missed, compared to the saturation of program crashes and regular code coverage, we demonstrate the superior accuracy of

our metric in reflecting fuzzing (cost-)efficiency. Parts of this chapter have appeared in a peer-reviewed publication [170], first-authored by the author of this thesis.

Chapter 7 – Vulnerability-Guided Fuzzing

This chapter presents a novel fuzzing approach that integrates our vulnerability-oriented coverage metric as the main test-optimization criterion into a directed fuzzer to target potentially vulnerable code. By comparing the effectiveness of our approach in detecting security bugs with coverage-based and existing vulnerability-guided fuzzing approaches, we demonstrate the superior accuracy of our metric in reflecting fuzzing effectiveness. Parts of this chapter have appeared in a submission [171], first-authored by the author of this thesis.

Chapter 8 – Related Work

This chapter presents related work on measuring fuzzing effectiveness, including existing (but inadequate) solutions for each sub-problem that had to be addressed for our solution approach. Parts of this chapter have appeared in a submission [171] and peer-reviewed publications [168–170], first-authored by the author of this thesis.

Chapter 9 – Conclusion

This chapter concludes this dissertation. It summarizes the main results and insights, discusses overarching limitations, and provides an outlook and ideas for future work.

Information

Several chapters of this dissertation are based on publications by the author of this thesis. These publications are referenced in the short abstracts at the beginning of each chapter. Due to the apparent content overlap, paraphrased excerpts from these publications within the respective chapters are not explicitly marked.

Contents

Part I.

Introduction and Background

1. Introduction

This chapter motivates the topic of this dissertation, outlines the research problem and gaps identified in the literature, and presents the solutions and contributions to fill them. Parts of this chapter have appeared in peer-reviewed publications [168–171], first-authored by the author of this thesis.

1.1. Context and Motivation

Software is ubiquitous today, touching almost every aspect of our personal and professional lives. While most software is fairly simple to use, its underlying architecture and functionality are often highly complex [197]. For instance, the operating systems on our PCs and smartphones, the browsers we surf the web with, and the social media platforms we use to stay in touch with family and friends—each of these examples consists of millions of lines of code (LoC) spread across hundreds of dependent software components such as libraries, frameworks, and (distributed) services [72, 145, 226].

However, a vulnerability in just one component can compromise the security of the entire software system [19, 184, 288], often resulting in severe financial and reputational damage if exploited by an attacker. One of the most prominent examples in this context is the *Heartbleed bug* [75], which was discovered in the OpenSSL cryptography library in 2014. A buffer over-read bug in the software could allow an attacker to extract sensitive data from the targeted (remote) server, opening the door to critical data leakage. Since the OpenSSL library is used in thousands of software products, this vulnerability affected millions of devices and caused an estimated $500 million in damage [227].

Such software bugs are not uncommon! The rapid pace of modern software development introduces new security bugs at a high rate, especially in software written in bug-prone programming languages such as C/C++ [209, 248, 293]. According to Zhu and Böhme [336], more than three-quarters (77%) of bugs in widely used open-source software (OSS) are due to recent code changes, as they found by analyzing some 23,000 C/C++ bug reports. This constant bug rate, along with increasing digitization, contributes to the rising cost of cybercrime, which in the US alone is estimated to grow from about $450 billion in 2024 to $1.8 trillion in 2028 [232].

To catch security bugs as early as possible, comprehensive and rigorous *software security testing* [84] must be performed as part of the development cycle [266]. This involves systematically examining and evaluating the system under test (SUT) to identify vulnerabilities, thus ensuring that it defends against threats and malicious attacks [14]. Thereby, software (security) testing is typically automated by tools that enable continuous testing of rapidly evolving software [76]. One such automated testing technique that has proven very powerful in finding security bugs in practice is *fuzz testing*.

1.1.1. Finding Software Bugs with Fuzzing

Fuzz testing, or *fuzzing* for short, was first introduced by Professor Barton P. Miller in 1990 [191] as a form of *random testing* [57, 109]. The SUT is repeatedly executed with randomized, malformed, and thus unexpected test data (*fuzz inputs*) to trigger program crashes [284, pp. 21–29]. These crashes indicate software bugs that violate the *dependability*, i.e., reliability, availability, and security [14], of the software. In C/C++ programs, the execution is thereby aborted either by the operating system or the selected (memory) sanitizer [274] once an unsafe program state is encountered—for example, due to memory corruption, such as an invalid memory access, buffer overflow, or memory leak [285, pp. 42–49, 261]. To facilitate a high throughput of fuzz inputs, the fuzzing process is automated by a *fuzzer*, which is connected to the SUT and then run for a certain period of time (*fuzzing campaign*) [177].

Fuzzing has become the dominant security testing technique [228], primarily due to the absence of false positives since every crash directly indicates a security bug. Therefore, it is less susceptible to the *state explosion problem* in real-world software than static analysis approaches [112], discovering thousands of bugs in popular OSS [99, 100, 240, 286, 326]. At Microsoft, their internally developed fuzzer "found roughly one third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide" [96]. Similarly, Meta reported that "fuzzing identified more than 1,000 bugs" [178] in critical software services. Google uses fuzzing not only to secure its own products, such as the Chrome browser [196], but also dependent third-party OSS [264].

1.1.2. Importance of Measuring Fuzzing Performance

To assess the risk of missed security bugs after fuzzing, it is important to accurately measure the performance of a fuzzing campaign—i.e., its potential to find all such bugs (*effectiveness*) with the available time or hardware resources (*efficiency*) [285, pp. 71–77]. However, measures that rely on bug manifestations can lead to incorrect assessments [98, 144, 309], because a campaign may appear highly performant by quickly detecting one or two bugs, while other bugs (e.g., in unreached parts of the program) remain undetected. This becomes even more challenging when no bugs are found, which means that either there are no bugs in the SUT or there are, but the campaign missed them. Although this question cannot be fully answered by software testing [68], indirect indicators (that are independent of bug manifestation), hereafter called *(proxy) metrics*, can be used to approximate the bug-finding probability of a campaign [308, 285, pp. 120–133]. Moreover, such metrics can be used to improve fuzzing since *test-evaluation criteria can also optimize the test generation* [122, 247, 234, pp. 151–159].

Evaluating Fuzzing Performance

From an analytical perspective, performance metrics allow for a more informed decision about when to (i) stop a single campaign (*stopping criterion*) or (ii) end fuzzing altogether

(*test-end criterion*) [284, pp. 437–470, 285, pp. 71–77, 23]. They can also be used to benchmark fuzzers [144, 189, 301], which, however, is not the focus of this dissertation.

Regarding (i), metric-based stopping criteria, as opposed to using a fixed fuzzing time budget (e.g., 24 hours), allow campaigns to be terminated dynamically, e.g., when their effectiveness saturates over an extended period. On the one hand, ineffective campaigns can thus be identified and stopped much earlier, saving valuable testing resources. On the other hand, effective campaigns that are close to bug detection can be prevented from being stopped prematurely with such dynamic criteria.

Regarding (ii), such metrics provide useful insights into the performance of the overall fuzzing efforts and, further, the dependability of the SUT. This is important not only to avoid unnecessary fuzzing costs but also to reduce the risk of missing bugs, e.g., by identifying untested or inadequately tested program parts (i.e., test gaps).

Enhancing Fuzzing Performance

From a constructive perspective, performance metrics allow for improving the bug-finding effectiveness and efficiency of fuzzing [177, 299]. For instance, in *mutation-based fuzzing* [24, 25, 326], such metrics determine which inputs are retained for further mutations (*optimization criterion*), thus increasing the campaign's chances of detecting security bugs over time. Ultimately, this helps to uncover critical vulnerabilities before potential attackers find and exploit them.

1.1.3. Inadequacy of Code Coverage in the Context of Fuzzing

The quality of the metric, i.e., its correlation strength with bug detection [31, 51], determines the metric's accuracy in reflecting fuzzing performance [87]. Hence, the more accurate the metric, the more efficient and effective fuzzing will be—whether employed as a stopping/test-end criterion [23, 170] or an optimization criterion [299].

The most preferred metric in software (security) testing is *code coverage* [26, 101, 130, 142, 189, 321], which measures the ratio of code regions (e.g., lines, basic blocks, or functions) in the SUT executed at least once by the test inputs [234, pp. 211–233]. As a result, higher code coverage increases the bug-finding probability since a bug can only be found if the corresponding faulty line is reached[1] (i.e., fewer test gaps). However, in the context of fuzzing, the expressiveness of code coverage is limited due to the typically *low coverage* of fuzzing campaigns [192, 325, 218, pp. 94–97], along with the *scarcity* [211, 339] and *dense concentration* of security bugs [172]. Accordingly, campaigns with high(er) code coverage may still miss the few bug-prone regions, while those with low(er) coverage may manage to execute them. Hence, by treating all code as equally important and, therefore, equally vulnerable, code coverage tends to *falsely approximate* the performance of fuzzing, rendering the development of better metrics a top priority in current fuzzing research [21].

[1] Note that code coverage alone, while necessary for bug detection, is not sufficient because the test data may not trigger the underlying bug despite execution [129].

1.1.4. Research Goal and Scope

The overarching goal of this dissertation is to develop a novel (proxy) metric that reflects the performance of fuzzing more accurately than code coverage. This includes the following two sub-goals, which also define the scope of this dissertation:

Goal 1: Demonstrate improved reflection of fuzzing efficiency, i.e., it allows saving more time or hardware resources while maintaining the bug-finding effectiveness.

Goal 2: Demonstrate improved reflection of fuzzing effectiveness, i.e., it allows detecting more bugs with the same available resources.

1.2. Problem and Research Gaps

Code coverage treats all code as equally important and, therefore, tends to falsely approximate fuzzing performance, which can have far-reaching practical consequences (see Section 1.1.2 on page 4). For this reason, this dissertation addresses the problem of *refining code coverage to accurately reflect fuzzing efficiency and effectiveness by effectively narrowing the focus to only the (subset of) potentially vulnerable code*. This requires (i) researching techniques to reliably identify bug-prone code and (ii) demonstrating the superiority of the resulting vulnerability-oriented code coverage metric by achieving Goal 1 and Goal 2. Related to these two sub-problems, we identify the following five research gaps:

Problem 1: Identifying potentially vulnerable source code.

Gap 1: **SAST Tool Evaluation.** Static application security testing (SAST) tools [52] are widely used in practice [17] for their ability to quickly scan the entire source code for security-critical regions. Although false positives (i.e., wrong SAST tool findings) are a known [56, 127, 135] and well-studied problem [5, 103, 137, 139, 204, 254, 294] of such tools, false negatives (i.e., missed bugs within the analyzer's detection scope) are a less-studied limitation, especially in the C/C++ domain. Existing studies on benchmarking static analyzers [10, 46, 94, 102, 126, 138, 219, 295] mainly employ artificial datasets with easy-to-find bugs introduced by simple syntactical code changes, thereby reporting detection rates of 80% and higher. However, it is unclear whether these success rates extend to the more complex security bugs in real-world software [36, 93], which is essential for reliably narrowing down the code scope for our metric.

Gap 2: **ML-Based Vulnerability Prediction.** In researching Gap 1, we found that SAST tools overlook many real-world vulnerabilities [168]. To mitigate this, experts recommend adapting such tools to the targeted codebase by fine-tuning their checks [83, 221, 290, 320], as well as complementing SAST with secure coding [261, pp. 505–510, 220] or code reviews [16, 29, 45, 111]. However, such optimizations require in-depth domain knowledge and manual effort, eventually limiting the practicality of our metric. Alternatively, various vulnerability heuristics based on code complexity [73, 187, 268, 337], code change rate [207, 268, 336], or text/bug-pattern mining [140, 297, 317] have

been proposed to effectively detect certain bugs. However, combining SAST tools findings with such heuristics to reliably detect a wide range of security bugs has barely been explored so far.

Problem 2: Demonstrating the improved accuracy of our metric in reflecting fuzzing efficiency (Gap 4 ⇔ Goal 1) and effectiveness (Gap 5 ⇔ Goal 2).

Gap 3: Real-Time Coverage Tracking. Collecting fuzzer-independent code coverage information in parallel with the fuzzing campaign is crucial for stopping campaigns dynamically [284, pp. 437–470]. The most common approach [2, 164, 189, 190] involves sampling a representative subset of all generated fuzz inputs and then replaying them on a second SUT instance compiled with coverage tracking [1, 239]. However, this approach is limited to fuzzers that persist performance-enhancing inputs, e.g., in a special directory, such as mutation-based fuzzers. Nonetheless, replaying fuzz inputs in real-time to monitor and stop campaigns adds significant hardware and synchronization overhead, making it impractical for large-scale fuzzing. Thus, a technical solution to these problems is lacking in fuzzing research.

Gap 4: Vulnerability-Oriented Fuzzing Stopping Criterion. An under-explored aspect of fuzzing research involves criteria for deciding when to stop a campaign. Currently, campaigns are stopped when (i) a fixed fuzzing time budget has expired [144, 177, 258], (ii) no progress in crash generation or code coverage is observed for an extended period [284, pp. 437–470, 285, pp. 71–77, 287], or (iii) the bug-finding probability falls below a certain threshold [23]. However, the metrics used in (i) and (ii) can exaggerate fuzzing effectiveness (see Section 1.1.3 on page 5), keeping campaigns alive even though they only trigger redundant crashes or cover non-critical code. Existing approaches for (iii) are fuzzer-dependent as they must account for the *adaptive bias* [20, 23] inherent in all fuzzing strategies where the probability of generating a bug-revealing fuzz input increases as the campaign progresses. Thus, more (cost-)efficient, fuzzer-independent stopping criteria (e.g., based on our metric) remain to be researched.

Gap 5: Vulnerability-Guided Fuzzing. Current fuzzing strategies primarily try to maximize overall code coverage in the SUT [25, 53, 86, 99, 157, 175, 235, 240, 326], with newer vulnerability-guided approaches [306] targeting complex [73], memory-accessing [305], or sanitizer-instrumented code [222, 265, 334]. However, due to the limitations discussed in Section 1.1.3 on page 5, coverage-based fuzzing has shown to be prohibitively expensive, requiring an exponential increase in time or hardware resources to achieve only a linear increase in bug detection [22]. Current vulnerability-guided fuzzing approaches are likely to run into similar problems as the imprecise vulnerability predictors they employ flag a lot of code and, therefore, again necessitate extensive code coverage. Thus, more effective fuzzing approaches (e.g., based on our metric) need to be investigated.

1.3. Solution

This dissertation proposes to improve code coverage by narrowing the focus to the SAST-flagged, potentially vulnerable code. We use our metric to tackle open research problems, thereby analyzing its predictive power through empirical evaluations.

Generally, we have opted for advanced SAST tools to identify problematic code. To understand the strengths and weaknesses of such tools, we examine the effectiveness of state-of-the-art analyzers in finding known real-world security bugs (⇒ addressing **Gap 1**). Based on the gained insights, we combine SAST with established code complexity measures into a machine learning (ML) model [71] to find bug-prone code more reliably (i.e., with fewer false negatives) than with static analyzers alone (⇒ addressing **Gap 2**).

To terminate fuzzing campaigns more cost-efficiently, we employ our metric as a saturation-based stopping criterion (⇒ addressing **Gap 4**). Specifically, before fuzzing, we flag the potentially vulnerable code identified by the ML vulnerability prediction model. During the campaign, we then use a new (real-time) coverage analyzer (⇒ addressing **Gap 3**) to track the progress of the flagged and covered code and further determine the campaign's termination point.

To detect bugs more effectively, we apply our metric as an optimization criterion in a (directed) fuzzer (⇒ addressing **Gap 5**), thereby using SAST tools to flag the bug-prone code. Moreover, our SAST-guided fuzzing approach dynamically disables likely false-positive and unreachable analyzer findings to mitigate their impact on the fuzzing performance. It also switches between vulnerability-guided and coverage-based fuzzing to account for bugs not flagged by the SAST tool (i.e., false negatives).

1.4. Contributions

By filling the identified research gaps, this dissertation makes the following contributions:

- ⋆ To fill **Gap 1**, we contribute an empirical study that evaluates the effectiveness of five free and one commercial SAST tool(s) in finding known real-world security bugs using a dataset built from 192 Common Vulnerabilities and Exposures (CVE) reports. Our results show that state-of-the-art static analyzers miss many bugs, with the best-performing tool identifying only about half (52.6%) of the bugs. Combining multiple tools can improve the detection rate by 10.9 to 20.3 percentage points (pp) compared to using a single analyzer; however, at the cost of flagging 14.6pp more functions [168].

- ⋆ To fill **Gap 2**, we contribute a qualitative analysis of 25 vulnerability indicators derived from SAST tool findings and code complexity metrics, which we combine into different ML vulnerability prediction models to reliably identify potentially vulnerable functions. Our evaluation shows that most of the trained models effectively discriminate between vulnerable and (likely) non-vulnerable functions in real-world programs with known security bugs, achieving true and false positive rates above 0.9 and below 0.5, respectively [170].

- ⋆ To fill **Gap 3**, we contribute a fuzzer-independent coverage analyzer that supports real-time tracking of code coverage of all fuzz inputs generated during a fuzzing campaign, useful for dynamically stopping campaigns. As part of this contribution, we provide a fuzzer coverage dataset (created with our analyzer) containing extensive code coverage and bug data from 9 fuzzers launched on 12 subject programs, worth over 12 CPU-years of fuzzing. This multipurpose dataset can also be used for fuzzing research outside of this dissertation [169].
- ⋆ To fill **Gap 4**, we contribute a vulnerability-oriented stopping criterion based on the saturation of our fuzzing metric. Our evaluation shows that, compared to saturation of triggered program crashes and regular code coverage, stopping 24-hour fuzzing campaigns with our criterion can save, on average, 6 to 12 hours of fuzzing time while missing less than 0.5 out of 12.5 bugs. Thus, these results demonstrate the improved accuracy of our metric in reflecting fuzzing (cost-)efficiency (⇒ achieving Goal 1) [170].
- ⋆ To fill **Gap 5**, we contribute a vulnerability-guided fuzzing approach that optimizes the test generation according to our metric. Our evaluation shows that our dynamic scheduling of fuzzing targets (i.e., SAST-flagged lines) leads to a higher target coverage than directed fuzzing with static targets. Moreover, compared to coverage-based fuzzing, our fuzzing approach detects 10 to 14 additional previously unknown security bugs with 3 to 4.5 times more different ones. Compared to the best vulnerability-guided baseline approaches, it detects 5 to 12 additional bugs, with 1.6 to 3 times more different ones. Thus, these results demonstrate the improved accuracy of our metric in reflecting fuzzing effectiveness (⇒ achieving Goal 2). Additionally, our approach finds the bugs in most subject programs, on average, up to 10.5 and 5.8 hours earlier during 48-hour campaigns than the fastest coverage-based and vulnerability-guided fuzzing baselines, respectively [171].

Parts of the above contributions have previously appeared in the peer-reviewed publications listed below. Figure 1.1 on the next page gives an overview of the research gaps and associated publications, grouped into the two sub-problems.

- P1 **Stephan Lipp**, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2022, pp. 544–555. DOI: 10.1145/3533767.3534380.
- P2 **Stephan Lipp**, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. "FuzzTastic: A Fine-grained, Fuzzer-agnostic Coverage Analyzer". In: *Proceedings of the International Conference on Software Engineering Companion*. ACM. 2022, pp. 75–79. DOI: 10.1145/3510454.3516847.
- P3 **Stephan Lipp**, Daniel Elsner, Severin Kacianka, Alexander Pretschner, Marcel Böhme, and Sebastian Banescu. "Green Fuzzing: A Saturation-Based Stopping Criterion using Vulnerability Prediction". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2023, pp. 127–139. DOI: 10.1145/3597926.3598043.

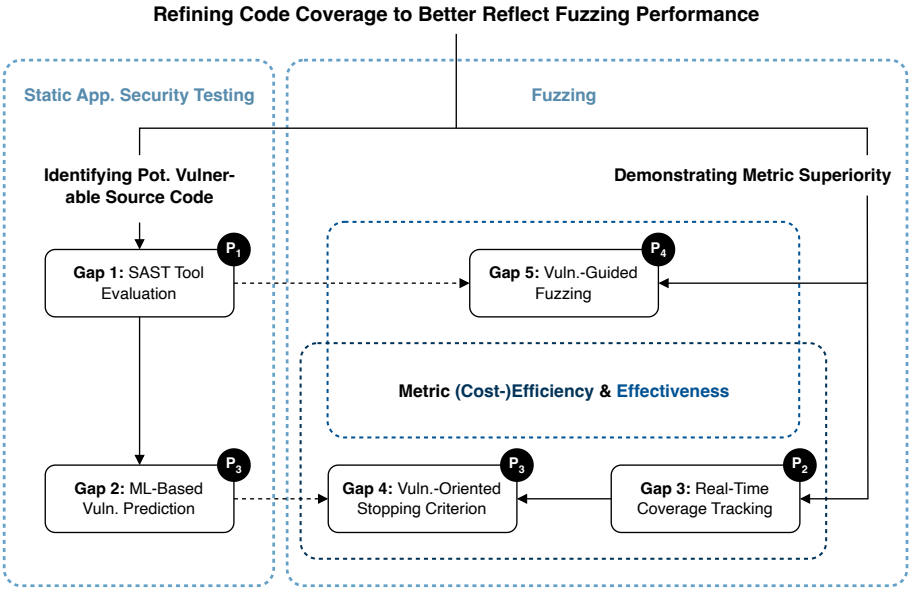


Figure 1.1.: Big picture of the research gaps and associated publications on which parts of this dissertation's chapters are based.

P4 **Stephan Lipp**, Severin Kacianka, Alexander Pretschner, and Marcel Böhme. "SAST-Guided Greybox Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. Under submission. ACM. 2026.

In addition to these papers, the author of this thesis has co-authored the following peer-reviewed publications that tackle related research problems but are not included in this dissertation:

- Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and **Stephan Lipp**. "Reachable Coverage: Estimating Saturation in Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2023, pp. 371–383. DOI: 10.1109/ICSE48619.2023.00042.
- Keno Hassler, Philipp Görz, **Stephan Lipp**, Thorsten Holz, and Marcel Böhme. "A Comparative Study of Fuzzers and Static Analysis Tools for Finding Memory Unsafety". *ACM Transactions on Software Engineering and Methodology* (2025). Under submission.
- Hoang Lam Nguyen, **Stephan Lipp**, Marcel Böhme, and Lars Grunske. "On the Impact of Ties in Bug-Based Fuzzer Benchmarking". In: *Proceedings of the International Conference on Software Engineering*. Under submission. ACM. 2026.

2. Background

This chapter provides the background knowledge required to understand the subsequent chapters. It covers the basic concepts, methodological and technical details, and quality criteria for static and predictive program analysis, as well as fuzzing. Parts of this chapter have appeared in peer-reviewed publications [168, 170, 171], first-authored by the author of this thesis.

2.1. Static and Predictive Bug Detection

In this section, we introduce the static and predictive code analysis techniques employed in this thesis to identify potentially vulnerable code. We first discuss their general application, then the various relevant sub-variants of each analysis technique, along with the quality metrics used to evaluate their effectiveness.

2.1.1. Static Application Security Testing

Static application security testing (SAST) is a software testing method that analyzes the source code (or intermediate or binary code) of the SUT *without* actually executing it [52]. It is basically an automated code review performed by a software tool, often called a SAST tool or static code analyzer, to identify potential security vulnerabilities early in the development process [54]. Hence, by using SAST, critical bugs in the code can be identified and fixed before they reach the final product, contributing to more secure and reliable software.

Workflow

To perform SAST, first, a static analyzer must be selected that supports the SUT programming language(s) and the types of vulnerabilities to be detected. Once a suitable analyzer has been found, it needs to be integrated into the development environment. This involves configuring the SUT for seamless, push-button analysis as well as adapting the SAST tool to the targeted codebase for the best possible bug-finding effectiveness.

The static analysis workflow[1] of a SAST tool is shown in Figure 2.1 on the following page. Here, the analyzer first transforms the source code of the SUT into a more abstract code representation, which it then examines for potential vulnerabilities. The level of abstraction directly affects the tradeoff between efficiency and effectiveness—a more detailed abstraction generally leads to an increased bug-finding effectiveness but a decreased efficiency, and vice versa. Hence, different tools employ different strategies, enabling the

[1] This process is initiated either manually by a software or security engineer or automatically via a continuous integration (CI) pipeline, e.g., after each (major) code change.

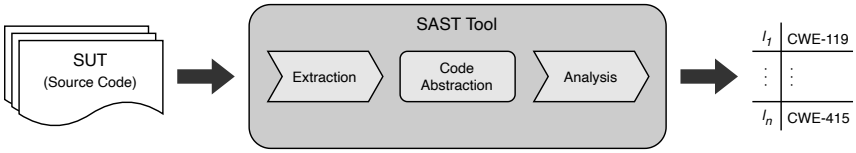


Figure 2.1.: Workflow of a static application security testing (SAST) tool, in which the analyzer, based on the scanned source code of the system under test (SUT), outputs the code locations it considers potentially vulnerable, along with the identified vulnerability type (represented by the corresponding CWE identifier).

application of various analysis techniques based on the chosen code abstraction (discussed in the next section).

During the analysis, the SAST tool scans the code for potentially vulnerable code based on its integrated security checks. Once this process is complete, it generates a report that, for each identified vulnerability, includes, among others, the code location (i.e., source file plus line) and the associated vulnerability type. Although the naming conventions for these types can vary across tools[2], many adhere to the standardized Common Weakness Enumeration (CWE) framework [194] for classifying software (and hardware) weaknesses.

While modern analyzers often support customization by adding new checks, their default checks for C/C++ typically focus on memory-related vulnerabilities, such as out-of-bounds memory accesses (CWE-119) and double-free errors (CWE-415).

Formally, a SAST tool run can then be described as follows, which becomes relevant when later defining our fuzzing metric.

Definition 2.1 (SAST Tool Analysis)**.** The static analysis of a SAST tool $\mathrm{t} \in \mathfrak{T}$ on the code regions $R \in \mathcal{R}$ can be described by the function $\texttt{analyze} : \mathfrak{T} \times \mathcal{R} \to \mathcal{R}$, which is implemented as follows:

$$\texttt{analyze}(\mathrm{t}, R) \coloneqq \{r \mid r \in R \text{ and } r \text{ contains lines flagged by } \mathrm{t}\}$$

In Definition 2.1, given a SAST tool and the set of code regions R in SUT to be examined, the analysis function returns the subset of examined regions $R' \subseteq R$ that contain one or more code locations flagged as potentially vulnerable by the tool.

The flagged code regions should then be inspected and, if they actually contain vulnerabilities, adequately fixed. After remediation, further testing should be performed to confirm that the vulnerabilities have been adequately addressed and, furthermore, to ensure that no new vulnerabilities have been introduced in the process. In this thesis, however, we do not perform this last step because the flagged, potentially vulnerable code is directly employed to assess or improve fuzzing.

[2]This variation also extends to the report data format. Newer SAST tools, however, primarily use the Static Analysis Results Interchange Format (SARIF) format [59], a standardized, tool-agnostic output format based on JavaScript Object Notation (JSON) that facilitates the integration of multiple analyzers.

Types of SAST

In general, SAST techniques can be divided into *syntactic* and *semantic* static analysis, each offering distinct strengths and weaknesses in terms of bug finding [168]. To provide a more comprehensive analysis, modern SAST tools often integrate several techniques from both categories.

Syntactic Static Analysis. Techniques in this category focus on vulnerabilities that can be detected by examining only the *syntax* of the SUT's source code for suspicious code patterns that indicate security bugs.

Pattern-Based Analysis: This technique scans the source code for predefined patterns associated with specific security vulnerabilities. These patterns can include code instructions, function calls, or regular expressions that match known insecure coding practices. Pattern-based analysis enables rapid scanning of the entire codebase and individual code snippets (which need not be compilable at analysis time), thus allowing for early detection of security bugs. However, because this technique ignores the semantics of the code, it often produces wrong warnings or misses more complex bugs [136].

Semantic Static Analysis. Techniques in this category go beyond syntax to further improve the bug-finding effectiveness. Although generally limited by the problem of *undecidability* [151], they aim to understand the *semantics* of the source code by reasoning about its control and data flow, as well as the relationships between the contained variables, to detect more complex vulnerabilities.

Control-Flow Analysis: This technique examines the potential execution paths of SUT enforced by conditional statements, loops, and function calls, focusing on the control flow rather than the deeper semantics of individual operations. This makes it possible to detect, e.g., infinite loops that may manifest as denial-of-service (DoS) vulnerabilities. Although this technique is usually more effective than pure syntactic analyses, it can still struggle with large, complex software systems that contain deeply nested control structures. In such cases, the analysis may sacrifice SAST accuracy to handle the complexity, which may result in wrong warnings or even missed bugs [4].

Data-Flow Analysis: This technique traces the data flow from its source to various destinations in the SUT to identify security bugs. Thereby, it focuses on instances where potentially untrustworthy data (e.g., user input) flows into sensitive operations (e.g., memory allocation in C/C++) without sufficient validation or sanitization. This analysis can uncover complex vulnerabilities that require extensive reasoning across multiple code regions. However, it is computationally intensive and, like control-flow analysis, can struggle with complex code structures, potentially compromising its accuracy [66, 255].

Symbolic Execution: This technique simulates the execution of the SUT using symbolic values instead of concrete ones. Specifically, it translates the control-flow paths into logical constraints that contain such symbolic values, enabling the systematic exploration of different program paths and their symbolic states to detect subtle security bugs. Additionally, by solving constraints that indicate potential vulnerabilities, concrete test inputs

can be generated to show the existence of a bug, as well as for debugging and remediation purposes. However, like the other semantic techniques, symbolic execution is computationally expensive. In particular, the number of possible paths can grow exponentially in complex programs, making it impossible to thoroughly analyze all of them [41, 143].

Abstract Interpretation: Unlike symbolic execution, which is a (static) testing technique, abstract interpretation is a formal verification technique that aims for sound analysis results. It constructs a sound approximation (abstract model) of the SUT's behavior to exhaustively prove the absence of certain types of vulnerabilities across all possible execution paths—i.e., it *guarantees* to find all bug instances within its scope. Thereby, it can handle complex code structures, dynamic behaviors, and, therefore, deep semantic SUT properties that may be difficult for other SAST techniques to reason about. However, abstract interpretation can be computationally intensive, especially for large programs. It also requires specialized knowledge for configuration and result interpretation, which is why we did not include this technique in this thesis [61].

Evaluation Metrics

In the context of static and predictive vulnerability detection, we define *true positives* (TP), *false positives* (FP), *true negatives* (TN), and *false negatives* (FN) as follows:

- `TP`: Number of vulnerable code regions correctly identified as such.
- `FP`: Number of non-vulnerable code regions incorrectly identified as vulnerable.
- `TN`: Number of non-vulnerable code regions correctly identified as such.
- `FN`: Number of vulnerable code regions not identified as such.

For such vulnerability prediction tools to be usable for our fuzzing metric, they must reliably identify potentially vulnerable code regions while flagging as few (non-vulnerable) regions as possible. To evaluate this tradeoff on a dataset containing vulnerable programs with a total of N code regions, we use the true positive rate (TPR), aka *detection rate* or *recall*, and the flagged code rate (FCR):

$$TPR := \frac{TP}{TP + FN} \tag{2.1}$$

$$FCR := \frac{TP + FP}{N} \tag{2.2}$$

Equation (2.1) calculates the proportion of vulnerable code regions included in the dataset that are correctly identified as such by the vulnerability prediction tool under evaluation. Accordingly, a high TPR indicates an effective tool, which is crucial for our metric to enable meaningful fuzzing assessments. Complementary to this, Equation (2.2) calculates the proportion of regions flagged as problematic, regardless of their vulnerability status. Note that the FCR is more appropriate than the *precision* metric[3] in this tradeoff analysis because (i) the number of false positives is generally difficult to determine when using real-world subject programs due to undetected vulnerabilities, and (ii) a small

[3]Precision measures the proportion of correctly identified vulnerable code regions out of all flagged ones.

amount of flagged code is more relevant to our fuzzing metric than the actual SAST precision. A high FCR then suggests that the tool is likely to consider a large amount of non-vulnerable code as vulnerable, given the small number of vulnerabilities typically contained in programs [211, 339]. This would ultimately reduce the expressiveness of our metric, at worst, offering no improvement over regular code coverage.

2.1.2. ML-Based Vulnerability Prediction

Another approach to statically identify bug-prone code is to use machine learning (ML). Unlike SAST, which directly analyzes source code of the SUT or an abstraction thereof, *supervised* ML-based vulnerability prediction relies on labeled data[4] of known vulnerabilities in order to predict potential new vulnerabilities [78, 225]. Supervised ML thereby distinguishes between two learning techniques, *classification* and *regression*, that differ in the type of outcome they predict [33]. Classification predicts a categorical or discrete outcome by assigning the input to a particular class (e.g., cat, dog, or bird in image classification). In contrast, regression predicts a continuous outcome, i.e., a real number that can take any value within a certain range (e.g., temperature prediction). In this thesis, we use *binary classification* to predict whether a given code region is potentially vulnerable or non-vulnerable.

Workflow

To build such a ML vulnerability prediction model, it is first necessary to collect training data that is indicative of vulnerable code, which also involves labeling the contained code regions as either vulnerable or non-vulnerable. The predictive performance of the model thereby strongly depends on the quality of the training dataset, with better data generally leading to better predictions [71].

Next, based on the training data, a set of ML features[5] must be selected that capture the characteristics of potential vulnerabilities. For instance, these features can be derived from the source code or the version control system (VCS) history. Furthermore, an appropriate ML algorithm must be chosen to train and thus build the prediction model, depending on the prediction type and the training data distribution. During training, the algorithm tries to learn patterns in the labeled data that link the selected features with the contained vulnerabilities, which are then encoded in the ML model. After training, the model's effectiveness at finding vulnerable code should be evaluated on a separate testing dataset to ensure that it generalizes to new, unseen data. If the model performs poorly, it may need to be retrained using (i) additional data points, (ii) a more appropriate feature set, or (iii) a different training algorithm. Otherwise, if the model performs well, it can be used to predict potentially vulnerable code in programs outside the dataset by extracting the feature values and feeding them into the model [34].

[4] There are also *unsupervised* ML algorithms that can handle unlabeled data by identifying and clustering patterns in the data [33]; however, these are less commonly used for predicting software vulnerabilities.

[5] A feature is a single measurable property of the data that is used as input to train a ML model as well as to make predictions [71].

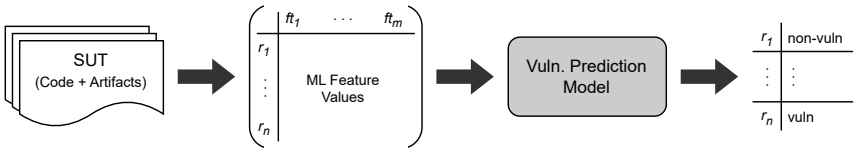


Figure 2.2.: Workflow of a machine learning (ML) vulnerability prediction model, in which the model predicts for each code region in the system under test (SUT) whether it is potentially vulnerable or not, based on the extracted feature values.

The vulnerability prediction workflow of a ML model is shown in Figure 2.2. Given the source code and possibly other relevant artifacts of the SUT, the values of the ML features used to train the model must first be extracted for each of the contained code regions. These feature values are then passed to the model, which in turn predicts the vulnerability status of each region.

Definition 2.2 (Cutoff θ)**.** A cutoff is a threshold value between 0 and 1 that determines the range of a model's predicted probability score at which a data point is assigned either the positive or negative output class.

Specifically, the model outputs a probability score for each code region, which is then converted into the corresponding class label based on a predefined *cutoff value* (see Definition 2.2). If the probability score is above or equal to the cutoff, the code region is classified as vulnerable (positive class). Otherwise, if the score is below the cutoff, the region is classified as non-vulnerable (negative class). While 0.5 is often used as the default cutoff, it can be adjusted to optimize the model for different objectives. Lowering the cutoff, for instance, increases the chances of identifying vulnerable code regions but may also flag more non-vulnerable regions (the preferred optimization for our metric). On the other hand, raising the cutoff may improve prediction accuracy by flagging fewer non-vulnerable regions but also increase the risk of missing vulnerable code.

Formally, the ML-based vulnerability prediction workflow for a single code region in the SUT can be described as follows.

Definition 2.3 (ML-Based Vulnerability Prediction)**.** The process in which a ML model $\mathfrak{m} \in \mathfrak{M}$ with the cutoff $\theta \in \mathbb{R}_{[0,1]}$ predicts the vulnerability status $L \coloneqq \{\mathit{vuln}, \mathit{non\text{-}vuln}\} \in \mathcal{L}$ (i.e., the classification labels) of a code region $r \in R$ based on its extracted feature values $X_r \in \mathcal{X}$ can be described by the function $\texttt{predict} : \mathfrak{M} \times \mathbb{R}_{[0,1]} \times \mathcal{X} \to \mathcal{L}$, which is implemented as follows:

$$\texttt{predict}\big(\mathfrak{m}, \theta, X_r\big) \coloneqq \begin{cases} \mathit{vuln}, & \text{if } \mathfrak{m}\big(X_r\big) \geq \theta, \\ \mathit{non\text{-}vuln}, & \text{otherwise.} \end{cases}$$

Note that Definition 2.3 describes the vulnerability prediction for a single code region. To extend this to all regions in the SUT, we adapt the function in Definition 2.1 on page 12 below to also support ML vulnerability prediction models.

Definition 2.4 (ML Vulnerability Prediction Model Analysis)**.** The vulnerability prediction of a ML model $\mathfrak{m} \in \mathfrak{M}$ with the cutoff $\theta \in \mathbb{R}_{[0,1]}$, denoted as $\mathfrak{m}_\theta$, on the code regions $R \in \mathcal{R}$ can be described by the function $\texttt{analyze} : \mathfrak{M} \times \mathcal{R} \to \mathcal{R}$, which is implemented as follows:

$$\texttt{analyze}\big(\mathfrak{m}_\theta, R\big) \coloneqq \Big\{ r \mid r \in R \wedge \texttt{predict}\big(\mathfrak{m}, \theta, X_r\big) = \mathit{vuln} \Big\}$$

The function in Definition 2.4 on the facing page returns the subset of analyzed code regions $R' \subseteq R$ that are classified as vulnerable by the employed ML model. Similar to the SAST workflow, the potentially vulnerable code regions should then be analyzed, remediated, and verified.

Evaluation Metrics

As with the SAST tools, the trained ML vulnerability prediction models must reliably identify bug-prone code regions while flagging as few (non-vulnerable) regions as possible. To evaluate this tradeoff, we use the true positive rate (TPR) (see Equation (2.1) on page 14) and the false positive rate (FPR), which is defined as follows:

$$\mathit{FPR} \coloneqq \frac{\mathit{FP}}{\mathit{FP} + \mathit{TN}} \tag{2.3}$$

Equation (2.3) calculates the proportion of non-vulnerable code regions that are incorrectly identified as vulnerable. To visualize the performance of a ML vulnerability prediction model during evaluation, we plot the TPR (y-axis) against the FPR (x-axis) at different cutoff values, resulting in a receiver operating characteristic (ROC) curve [81]. The ROC curve of a well-performing model then converges towards the top-left corner, indicating a high TPR with a low FPR. The area under the curve (AUC) of the ROC curve summarizes this performance as a single metric, with a higher ROC-AUC signifying better discrimination between the output classes.

2.2. Fuzz Testing aka Fuzzing

In this section, we start with an introduction to fuzzing, followed by an overview of different fuzzing types. We then focus on mutation-based greybox fuzzing, in particular, on commonly used optimization criteria for test generation. Finally, we outline current metrics used to evaluate the performance of fuzzing.

2.2.1. Overview

Fuzzing [177, 191] is a dynamic software testing technique that has proven highly effective in detecting security bugs [228], especially in C/C++ programs. The approach is straightforward: generate millions of random test inputs (aka *fuzz inputs*), feed them into the SUT, and monitor the program executions for crashes until a specified timeout is reached. This whole process is denoted a *fuzzing campaign* and is automated by a software tool called a *fuzzer*. Furthermore, crashes that could lead to malicious code execution, data

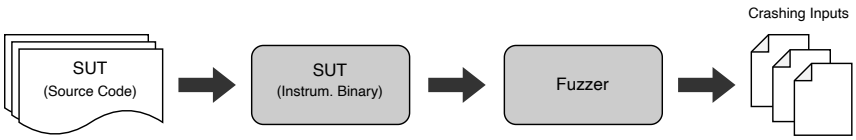


Figure 2.3.: Workflow of a fuzzer, in which the (instrumented) system under test (SUT) binary is executed with randomly generated test inputs to find inputs that crash the program execution and thus indicate security bugs.

leakage, or DoS are considered security vulnerabilities [285, pp. 42–54]. Thus, unlike static vulnerability detectors, fuzzing does not suffer from false positives, as each triggered crash directly indicates a software bug.

Workflow

To perform fuzzing, a suitable fuzzer must be selected for the targeted software, with many different fuzzers freely available online. Also note that when fuzzing libraries, a *fuzz harness* must be developed, i.e., a piece of code that connects the fuzzer to the SUT to enable fuzzing. Usually, several different harnesses are needed to thoroughly test the entire SUT.

The actual workflow of a fuzzer is shown in Figure 2.3. First, the source code of the SUT (along with that of the fuzz harness) must be compiled into a fuzzable binary. Depending on the chosen fuzzer, compile-time instrumentation is used to add instructions to the binary that, during test execution, (i) track the code regions covered by the fuzz inputs, (ii) measure their execution time to identify performance bottlenecks, and (iii) detect subtle errors that may not cause immediate crashes. The third point is known as *code sanitization*, which is supported by various error detectors (aka *sanitizers*) such as AddressSanitizer [263] for catching memory-related errors.

Once compiled, the instrumented binary is ready for fuzzing. For this, the fuzzer is attached to the binary, initiating the fuzzing campaign with a configuration that specifies, among others, the duration of the campaign, its resource limits to prevent exhaustion, and the model or corpus of valid sample inputs. During fuzzing, the instrumentation provides feedback to guide the fuzzer's test generation toward specific objectives (e.g., maximizing code coverage). When a program crash is observed, the corresponding fuzz input is stored in a dedicated directory.

After the campaign ended, the crash-inducing inputs should be analyzed to check if they reveal security vulnerabilities. The associated crash dumps, stack traces, and covered code regions can thereby help to identify, locate, and fix the underlying issue(s) in the code. Once the problem has been fixed, the fuzzer should be executed again (e.g., with the crashing inputs as seeds) to ensure that the bugs have been resolved.

Types of Fuzzing

In recent years, various fuzzing approaches have been developed that can be categorized along two dimensions: (i) the method of fuzz input generation (*generation-based* and

mutation-based) and (ii) the visibility level of the SUT's internals (*blackbox*, *whitebox*, and *greybox*) [177]. In the following, we describe each category in more detail.

Generation- and Mutation-Based Fuzzing. *Generation-based fuzzing* relies on a model of the SUT's input format, such as a template, constraints, grammar, or any other specification, in order to generate fuzz inputs that conform to the input structure [96, 284]. Hence, this approach is capable of producing diverse test inputs that explore the SUT more systematically and are, therefore, more likely to detect complex security bugs. This is particularly relevant for programs that require well-structured inputs to test code beyond the input parsing. However, creating a new input model can be very time-consuming and requires a deep understanding of the SUT's input format.

In contrast, *mutation-based fuzzing* starts the campaign with a set of input samples (*seed corpus*) accepted by the target program [177]. These seed inputs are repeatedly mutated during the campaign to derive new (semi-valid) fuzz inputs. The goal is to generate inputs that are valid enough to pass the initial parsing checks but invalid enough to exhibit unexpected behavior. Thus, this approach requires minimal knowledge of the SUT's input format, making it relatively easy to apply in practice. However, the quality of the employed seed corpus strongly affects the fuzzing effectiveness, with a low-quality corpus reducing the effectiveness.

Blackbox, Whitebox, and Greybox Fuzzing. Fuzzing began as blackbox *random testing* [191], checking only for external program crashes without monitoring the internal test executions after feeding the target program with pure random inputs. Despite early successes in finding security bugs, *blackbox fuzzing* usually struggles to overcome more complex conditions that would allow testing of deeper, more bug-prone code regions of the SUT, thus limiting its effectiveness [284, pp. 9–14, 177].

To reach untested code more effectively, *whitebox fuzzing* [39, 40, 97], aka dynamic symbolic execution or concolic (concrete + symbolic) testing, has been developed. It translates the control-flow paths of the SUT into logical constraints by concatenating the path conditions, which, when resolved, allow the derivation of concrete test inputs [58]. These inputs can then be executed directly by the target program or used as seeds for mutation-based fuzzing, as in *hybrid fuzzing* [176, 218, 224]. However, the involved constraint solving slows down the test generation and can also hinder the scalability to larger programs due to the exponential increase in paths to be analyzed [41, 260].

Greybox fuzzing addresses these limitations by combining elements of blackbox and whitebox fuzzing [177]. It uses lightweight program instrumentation to obtain feedback about the test executions, which is then used to improve the test generation throughout the campaign. The most common form is mutation-based greybox fuzzing [24, 25, 326], which utilizes runtime information like code coverage to guide the generation of new fuzz inputs by selectively mutating inputs that are promising according to the employed optimization criteria. Thus, this approach combines the simplicity of mutation-based fuzzing with the guidance of feedback-driven exploration, making it very effective at detecting security bugs.

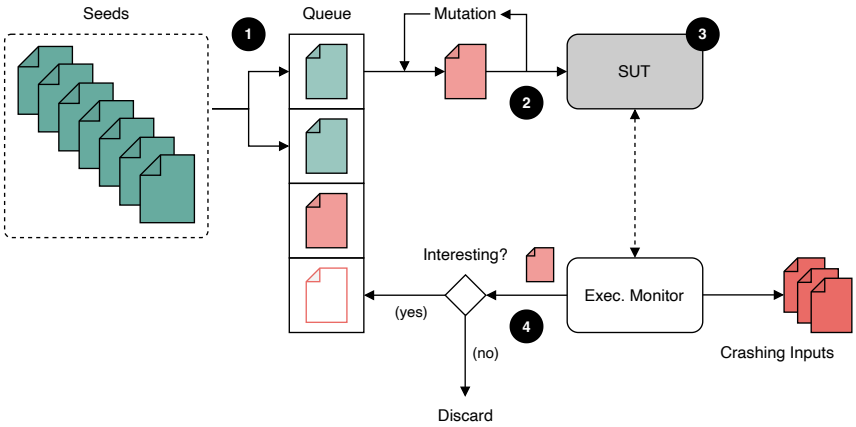


Figure 2.4.: Big picture of mutation-based greybox fuzzing, from seed selection and trimming to test generation, execution, and monitoring.

2.2.2. Mutation-Based Greybox Fuzzing

Mutation-based greybox fuzzing is the most commonly used fuzzing approach [228]. Below, we explain its internal functioning as well as the two main optimization criteria that guide the generation of fuzz inputs.

Approach

Figure 2.4 provides an overview of mutation-based greybox fuzzing with a specific test-optimization criterion. In this figure, we assume that the target binary is already properly instrumented.

In **step** ❶, each input in the seed corpus is executed at least once in the target program to (i) trim unnecessary bytes from large seed inputs (while preserving their execution behavior) to ensure high test throughput, and (ii) filter out seeds with similar behavior to increase the diversity of fuzz inputs generated later. The trimmed, unique seeds are then loaded into the mutation queue.

In **step** ❷, the top input in the queue is selected, followed by applying various mutations to its data. For instance, at the byte level, these mutations can be simple bit flips (i.e., changing 0s to 1s and vice versa), byte modifications (i.e., adding, removing, or replacing bytes with random values or specific bit patterns), or block swaps (i.e., rearranging blocks of data). At the string level, they can involve changing values (i.e., increasing, decreasing, maximizing, or minimizing them) or using dictionary-based mutations that replace parts of the input with predefined values or strings. Typically, a sequence of these mutation operators is applied to an input before passing it to the target program.

In **step** ❸, the mutated input is executed in the target program. The test execution is thereby monitored by a special software unit that interacts with the information provided by the code instrumentation in order to evaluate the fuzz input with respect to the employed optimization criterion. Also, in case of a program crash, the corresponding input is stored in a dedicated directory for later analysis.

In **step** ❹, the information collected so far by the fuzzing campaign is updated with that of the last test execution to calculate the new optimization score. If this score has improved compared to that of the previous executions, the campaign is getting closer to its optimization goal and, therefore, the fuzz input is added to the queue for further mutations. Otherwise, the input is discarded and will thus not be considered for future mutations. Steps 2 to 4 are then repeated until the campaign is stopped.

Optimization Criteria

In this section, we describe the two most commonly used optimization criteria for test generation in mutation-based greybox fuzzing. As mentioned earlier, the quality of an executed fuzz input, as measured by such a criterion, determines whether it should be discarded or kept for further mutations. Consequently, each newly enqueued seed input brings the fuzzing campaign closer to the optimization goal, which is why this approach is also referred to as *evolutionary fuzzing* [65].

Code Coverage. As the name suggests, this optimization criterion focuses on maximizing the amount of code covered (i.e., executed at least once) by the generated fuzz inputs. Accordingly, fuzz inputs that succeed in increasing code coverage are added to the queue for further mutations. This fuzzing approach is called *coverage-based fuzzing* [326].

An example of this is shown in Figure 2.5 on the next page, which depicts the inter-procedural control-flow graph (iCFG)[6], along with the execution traces and covered basic blocks (BBs)[7] (grey nodes) of the two fuzz inputs I_i (green trace) and I_{i+1} (blue trace) generated during the fuzzing campaign. Here, input I_i covers 6 of the 14 BBs, yielding a BB coverage of $6/14$ (43%). Assuming that I_i has not increased the total code coverage achieved so far, it will not be added to the queue. In contrast, I_{i+1} reaches two BBs that were previously not covered, even though achieving a lower BB coverage of only $4/14$ (29%). This increases the total BB coverage to $8/14$ (57%), thus qualifying I_{i+1} to be added to the queue so that mutations thereof may further increase the code coverage. This process is then repeated for each subsequent fuzz input $I_{i+2}, \ldots, I_n$.

Coverage of Specific Targets. Unlike the previous criterion, this one aims to minimize the distance between the inputs' execution trace and a subset of (potentially vulnerable) code locations, i.e., the so-called *fuzzing targets*. These targets are identified prior to fuzzing, either manually by security experts or automatically by vulnerability prediction tools. The goal is to direct the campaign towards these target locations in order to maximize target coverage, which is why this approach is referred to as *directed fuzzing* [24]. Thereby, directed greybox fuzzing generally consists of two phases: *exploration* and *exploitation*. The former phase aims to maximize code coverage in order to expand the seed corpus and avoid local optima, while the latter aims to reach the specified targets.

[6] A iCFG represents all possible execution traces in the entire SUT [4, 195, pp. 99–102].
[7] A BB denotes a linear sequence of instructions without branches [4].

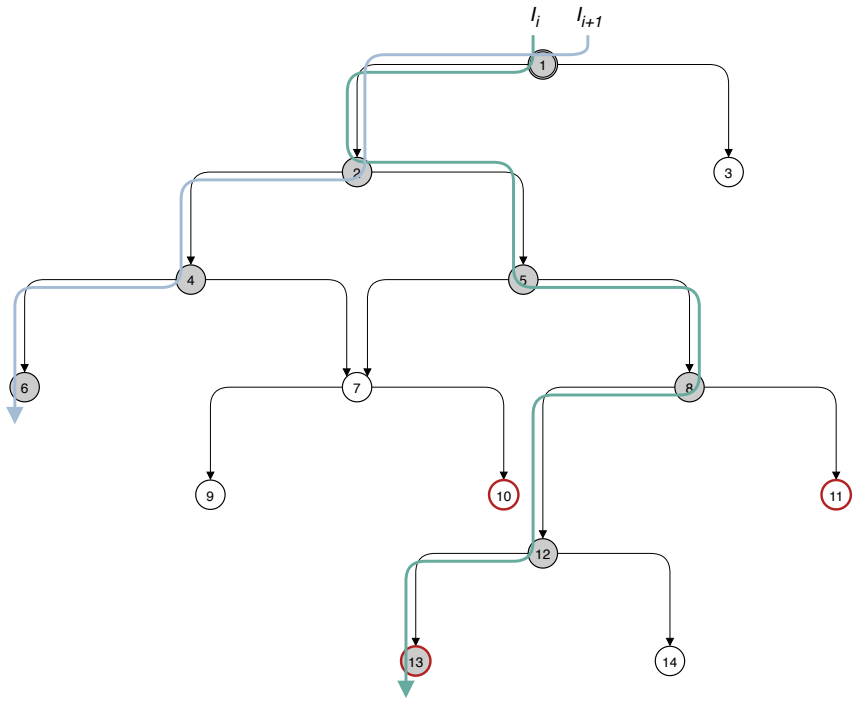


Figure 2.5.: Inter-procedural control-flow graph of an example program, where grey nodes indicate basic blocks (BBs) covered during the execution of the fuzz inputs I_i or I_{i+1}. Nodes outlined in red represent BBs that contain at least one potentially vulnerable line of code.

Note that by using the target distance rather than the target coverage as the optimization criterion, inputs that come closer to the targets are added to the queue, even if they do not yet reach them. This results in better guidance and, ultimately, higher target coverage.

At the BB level, the distance between two BBs is computed as the number of iCFG edges along the *shortest path* between them, as found by Dijkstra's algorithm [67]. The distance from one BB to several others is then calculated as the harmonic mean[8] of the distances from the source BB to each target BB [24]. However, if the source BB is among the target BBs, its distance is 0. Furthermore, if a target BB is (statically) not reachable from the source BB, it is not considered in the distance calculations.

To explain the distance calculations in more detail, we use the example[9] in Figure 2.5 again, in which the red nodes represent the BBs targeted during fuzzing. There, fuzz input I_i covers the BBs b_1, b_2, b_5, b_8, b_{12}, and b_{13}, each with a (harmonic mean) distance to the target BBs b_{10}, b_{11}, and b_{13} of:

- $d(b_1) = {}^{3}/_{1/4+1/4+1/5} = 4.29$
- $d(b_2) = {}^{3}/_{1/3+1/3+1/4} = 3.27$
- $d(b_5) = {}^{3}/_{1/2+1/2+1/3} = 2.25$

[8] The harmonic mean is less sensitive than the arithmetic mean to widely distributed or unreachable targets.
[9] We use the iCFG in this example for simplicity. However, most directed fuzzing implementations use the control- and call graph for the distance calculations [24].

```
==4122601==ERROR: AddressSanitizer: global-buffer-overflow on address [...]
READ of size 48 at [...] thread T0
    #0 [...] in __interceptor_memcpy %ASAN%/sanitizer_common_interceptors.inc:810:5
    #1 [...] in newStringLength %PKL%/src/core/value.c:304:36
    #2 [...] in varMultiply %PKL%/src/core/core.c:1775:21
    ...
    #5 [...] in main %PKL%/cli/main.c:132:23
    #6 [...] in __libc_start_main %GLIBC%/libc-start.c:308:16
    #7 [...] in _start (%PKL%/pocketlang)
```

Listing 2.1.: Example output of AddressSanitizer detecting a buffer-overflow vulnerability in Pocketlang. Besides the vulnerability type, it includes the stack trace of the test execution that triggered the vulnerability.

- $d(b_8) = {}^{2}/_{1/1+1/2} = 1.33$ (b_{10} is unreachable from b_8)
- $d(b_{12}) = {}^{1}/_{1/1} = 1$ (b_{10} and b_{11} are unreachable from b_{12})
- $d(b_{13}) = 0$ (b_{10} and b_{11} are unreachable from b_{13}, which is also a target BB)

The distance of I_i's execution trace τ_i is then calculated as the arithmetic mean of the distances of the covered BBs, i.e., $d(\tau_i) = {}^{d(b_1)+d(b_2)+d(b_5)+d(b_8)+d(b_{12})+d(b_{13})}/_{\mathbf{len}(\tau_i)} = {}^{4.29+3.27+2.25+1.33+1+0}/_{6} = 2.02$, where $\mathbf{len}(\cdot)$ returns the number of BBs covered in the trace. Assuming that I_i reduces the distance to the targets compared to the previous fuzz inputs, it will be added to the mutation queue. In contrast, the trace of input I_{i+1} has a distance of $d(\tau_{i+1}) = {}^{d(b_1)+d(b_2)+d(b_4)}/_{\mathbf{len}(\tau_{i+1})} = {}^{4.29+3.27+2}/_{3} = 3.19$ (no target BBs reachable from b_6), which is greater than that of τ_i, indicating that I_{i+1} is moving further away from the targets. As a result, I_{i+1} will not be added to the queue and thus will not be considered for future mutations. After that, this process continues for the subsequent inputs $I_{i+2}, \ldots, I_n$.

2.2.3. Fuzzing Metrics

Several metrics have been proposed to evaluate the effectiveness of fuzzing campaigns [285, pp. 120–133]. Below, we discuss the two most commonly used ones.

Number of Unique Bugs

The simplest way of measuring fuzzing effectiveness is to count the number of security bugs detected during the fuzzing campaign—the more bugs found, the more effective the campaign is deemed. However, it can be very challenging to accurately determine the number of unique bugs found because fuzzing only provides the generated fuzz inputs that cause the program crashes. Hence, multiple crashes can result from the same faulty instruction in the code [144], so the number of crash-inducing inputs does not necessarily equal the number of unique bugs.

To address this, the triggered crashes must be *deduplicated* into unique bugs by analyzing their root causes. Since a single campaign can produce thousands of crashes, manual

deduplication is very time-consuming and thus impractical in our case. Therefore, we perform an automated deduplication by clustering crashes based on the similarity of the stack frames in their execution traces [64]. Specifically, we follow ClusterFuzz's approach [11] and establish similarity by comparing the function names of the top $n \in \mathbb{N}^+$ stack frames and, if available, the vulnerability types reported by the sanitizer. The concrete value of n is thereby set based on the most accurate deduplication after manually examining a random subset of all crashes in our subject programs.

An example of such a stack trace is shown in Listing 2.1 on the previous page. It displays the called functions in the Pocketlang program up to `newStringLength`, in which a buffer overflow was intercepted by AddressSanitizer through its proxy function for the `memcpy` C library function. Now, when using $n = 3$ for deduplication, all crashes that trigger a "global-buffer-overflow" vulnerability and share the call hierarchy `__interceptor_memcpy` (frame #0), `newStringLength` (frame #1), and `varMultiply` (frame #2) would be grouped as instances of the same underlying bug.

Code Coverage

While the number of bugs can be useful for comparing the effectiveness of multiple campaigns relative to each other, it is difficult to interpret in absolute terms since we never know the total number of bugs contained in the SUT (see Section 1.1.2 on page 4). For this reason, code coverage is another widely used metric in the fuzzing domain, measuring the proportion of code regions in the SUT that have been executed at least once by the generated fuzz inputs [234, pp. 211–233]. Since a bug can only be detected if the corresponding faulty code is executed, the rationale is that a high code coverage increases the likelihood of discovering bugs and thus indicates an effective campaign.

Below, we define the code coverage of a campaign launched on a program with the code regions R. To check whether a region $r \in R$ has been covered within the first $t \in T$ time units of the campaign, we use the function $\texttt{covered} : R \times T \to \{\mathit{true}, \mathit{false}\}$, which is implemented as follows:

$$\texttt{covered}(r,t) := \begin{cases} \mathit{true}, & \text{if } r \text{ was covered at least once} \\ & \text{by a fuzz input within } t, \\ \mathit{false}, & \text{otherwise.} \end{cases} \tag{2.4}$$

Definition 2.5 (Code Coverage)**.** Using the function in Equation (2.4), the code coverage metric $S_{\text{cov}} : \mathcal{R} \times T \to \mathbb{R}_{[0,1]}$ is defined for the code regions $R \in \mathcal{R}$ in the SUT after $t \in T$ time units within the fuzzing campaign as follows:

$$S_{\text{cov}}(R,t) := \frac{\mathbf{card}\Big(\big\{r \mid r \in R \wedge \texttt{covered}(r,t)\big\}\Big)}{\mathbf{card}(R)}$$

where $\mathbf{card}(\cdot)$ returns the set cardinality.

Note that code coverage, as defined in Definition 2.5, can be measured at different code granularities, meaning that code regions can be lines, BBs, functions, or edges in the

control-flow graph (CFG). Although coverage alone does not guarantee that a bug will be triggered (as specific data is also required to exploit the faulty line), it is *necessary* to expose the bug. Hence, any bugs in an uncovered code region cannot be detected. For example, a 50% code coverage means that half of the SUT's source code has been executed while the other half remains untested, reducing our confidence that the SUT does not contain any obvious bugs.

Part II.

Identifying Potentially Vulnerable Source Code

3. SAST Tool Evaluation

This chapter empirically evaluates the effectiveness of several advanced static application security testing (SAST) tools in identifying potentially vulnerable code in real-world subject programs with known vulnerabilities. Parts of this chapter have appeared in a peer-reviewed publication [168], first-authored by the author of this thesis.

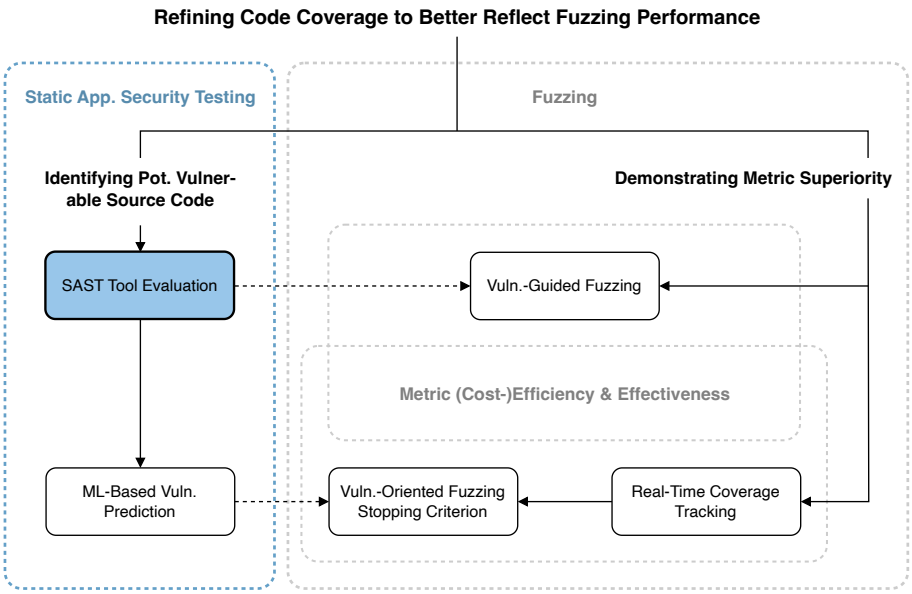


Figure 3.1.: Big Picture (Contribution 1: SAST Tool Evaluation).

3.1. Introduction

Context. As shown in Figure 3.1, this chapter belongs to the part of this dissertation where we explore approaches to identify potentially vulnerable code in the SUT. The coverage of the identified bug-prone code is then measured during fuzzing to estimate its performance. Hence, for our approach to work, the problematic code must be identified very reliably. We start our research with state-of-the-art SAST tools because they (i) are easy to set up and run [52, 110], (ii) can quickly scan large codebases [110, 171], (iii) support many C/C++ vulnerabilities [168], and (ix) are trusted by thousands of software projects [17, 110] and developers [54, 256, 294], suggesting that they are very useful for finding problematic code.

State-of-Practice and Problem. While the problem of wrong SAST tool findings (i.e., false positives) is well-researched [5, 56, 103, 127, 135, 137, 139, 204, 254, 294], there is a notable gap in analyzing their actual effectiveness in finding security bugs, especially in the context of C/C++. Here, existing studies mostly employ artificial datasets [10, 46, 94, 102, 126, 138, 219, 295], consisting of easy-to-find bugs. These are often simple syntactical code changes such as off-by-one array accesses that are introduced either automatically by bug-injection engines [70] or manually by security experts [27]. As a result, the cited papers report detection rates of around 80% for some of the static analyzers studied and even up to 100% for certain types of vulnerabilities. However, it remains unclear whether these success rates also *translate* to real-world software, which typically contains much more complex vulnerabilities [36, 93]. This caveat also calls into question the increase in bug detection reported by some of these papers when using multiple static analyzers in combination.

Solution Approach. To address this research gap, we evaluate the effectiveness of five free and one commercial SAST tools on 27 *real-world* software projects containing a total of 192 known vulnerabilities as documented in CVE reports, which serve as ground truth. We include a preliminary study, where we verify the general applicability of our CVE-based dataset for such an evaluation, thereby determining the function level as the optimal code granularity to approximate vulnerability detection. For this, we examine the type of code location (root-cause or bug-manifestation location) typically flagged by static analyzers and the type provided by the CVEs in our dataset. Any discrepancy between these locations could lead to incorrect results, as we cannot assess whether a vulnerability has actually been identified. In addition, we evaluate the tradeoff between improved bug detection and additional functions flagged (i.e., more false positives) when using multiple static analyzers together. We also perform this analysis separately for each vulnerability type to better understand the strengths and weaknesses of SAST.

Contributions. Overall, we contribute an empirical evaluation of the effectiveness of modern SAST tools in finding known real-world vulnerabilities (CVEs). Our results show that static analyzers miss many security bugs, with the best-performing tool identifying only about half (52.6%) of the bugs. Combining multiple tools can improve the detection rate by 10.9 to 20.3 percentage points (pp) compared to using a single analyzer; however, at the cost of flagging 14.6pp more functions.

3.2. Scope

In this section, we outline the scope of our evaluation. Specifically, we describe the selected SAST tools and subject programs, including the vulnerability types they contain.

3.2.1. SAST Tools

For our evaluation, we select the following SAST tools that support C/C++ and implement state-of-the-art SAST techniques. The first four static analyzers are open-source and freely

available, and—according to their GitHub stars [28]—are widely used in practice. CodeQL is free for academic purposes, although its analysis engine is proprietary. The last tool, CommSCA, is entirely proprietary and only commercially available.

Flawfinder (FLF): We use version 2.0.11 of this analyzer, which implements a syntactic SAST technique that scans the source code for potentially vulnerable code patterns stored in a local database. The built-in checks primarily identify functions that have been vulnerable in the past and also assess their security risk by analyzing the function arguments [310].

Cppcheck (CPC): We use version 2.3 of this analyzer, which implements a SAST technique that combines syntactic and semantic analyses to scan the code for potential security bugs. It uses a lightweight data-flow analysis to find bugs caused by undefined C/C++ behavior or dangerous code constructs [179].

Infer (IFR): We use version 0.14.0 of this analyzer, which is developed by Meta (formerly known as Facebook). It implements a semantic SAST technique based on *separation logic* [250] and *bi-adduction* [42], which allows Infer to reason about pointer structures (e.g., those in C/C++ to manipulate memory) and thus to detect, among others, subtle memory-safety security bugs [188].

CodeChecker (CCH): We use version 2.0 of this static analysis platform, which by default uses the analyzers Clang Static Analyzer [237] and Clang-Tidy [238], both part of the LLVM+Clang toolchain [153]. The former implements a semantic SAST technique based on *symbolic execution* [143], while the latter tool implements a syntactic technique based on *linting* [150]. Together, they support the most common C/C++ vulnerabilities [79].

CodeQL (CQL): We use version 2.1.3 of this analyzer, which implements a semantic SAST technique, where the analysis engine is decoupled from the actual (security) checks. CodeQL first translates the source code into a structured data representation, which is then queried using a specific language [13] to detect potential vulnerabilities. In our evaluation, we use the external CodeQL checks of version 1.23.1 [95].

CommSCA (CSA): This static analyzer is the only commercial tool that we include. To protect the company behind this analyzer, we anonymize its name and also do not reveal the implemented SAST technique.

We studied the documentation of each SAST tool, and if an implemented check for a vulnerability type included in our evaluation was not enabled by default, we enabled it. We also disabled checks that did not focus on security-related bugs, such as code smells or likely unreachable code.

3.2.2. Subject Programs

For our evaluation, we looked for real-world C/C++ programs with diverse and well-documented vulnerabilities. Unfortunately, such datasets are rather rare, and those available contain only very few vulnerabilities [173] or do not specify the vulnerability

Table 3.1.: Subject programs for the SAST tool evaluation (as appeared in [168]).

Subject	Version	# Lines	# Functions	# Vulns.
Binutils	2.29	134,767	4,071	59
FFmpeg	n3.3.2	413,353	17,788	22
Libpng	1.6.38	10,184	398	7
LibTIFF	4.1.0	19,527	826	13
Libxml2	2.9.10	85,442	2,982	17
OpenSSL	3.0.0	165,187	13,036	21
PHP	8.0.0-dev	209,407	9,145	15
Poppler	0.88.0	63,561	4,659	22
SQLite3	3.32.0	53,272	2,298	16
Total		1,154,700	55,203	192

types [7], which we need to check if an analyzer has correctly identified a security bug. One exception is Magma [114], a dataset built from 111 CVE reports. It uses a method called *front-porting*, which reintroduces publicly reported vulnerabilities into the latest version of the program. For each ported vulnerability, Magma also provides detailed information about its root cause, as well as where in the code the erroneous program state may manifest. Although Magma was created for fuzzer benchmarks, the vulnerabilities it contains should also be detectable by SAST tools.

In addition to Magma version v1.1, we include an older version of the Binutils suite, which consists of 19 programs for manipulating compiled code, and the FFmpeg multimedia processing tool. Together, they contribute 81 *non-front-ported* vulnerabilities. An overview of all the subject programs used in our evaluation is provided in Table 3.1. This table is complemented by Figure 3.2 on the next page, which shows that for all subjects except SQLite3, the average LoC of the vulnerable functions is below 100 lines.

Since many of the analyzers need to be attached to the build process in order to scan the SUT, we verified that none of the vulnerable code had been removed by the preprocessor (e.g., due to an improper build configuration). In cases where vulnerable code was removed, we either reconfigured the build process or, if this was not possible, omitted the vulnerability from the evaluation. Apart from this, we modified the SQLite3 build process so that all source files are compiled (and analyzed) separately rather than merging[1] all files into a single file before compilation. This ensures that findings from SAST tools that attach to the build process can be accurately attributed to the original source lines, which is mandatory to verify if a vulnerability has been found. Furthermore, since the Binutils vulnerabilities are spread across 19 different programs, as well as each program's code for manipulating different target binary formats, we compiled and analyzed the Binutils programs separately for each of the affected binary formats. Thereby, we ensured that vulnerable code shared by multiple Binutils programs is not wrongly counted more than once in our evaluation.

[1] `https://sqlite.org/amalgamation.html`

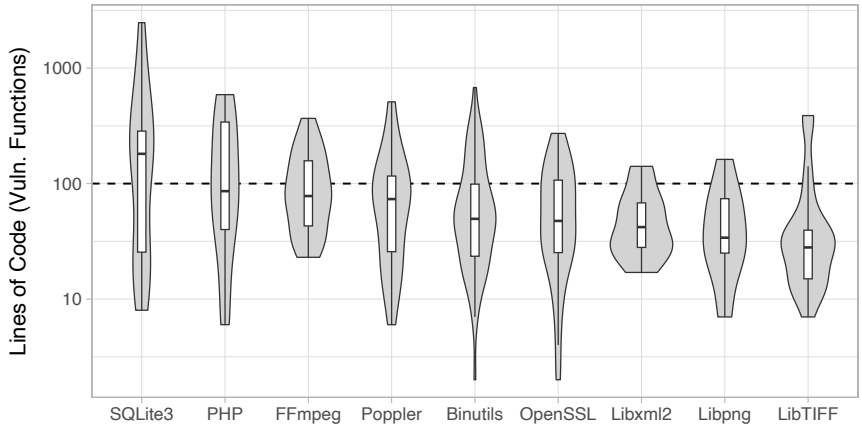


Figure 3.2.: Log-scaled distribution of lines of code of the functions in the different subject programs that are affected by at least one vulnerability (as appeared in [168]). The dashed line indicates the 100-line threshold.

Vulnerability Types and Classes

To distinguish between different types of vulnerabilities, we use the CWE [194], a widely recognized system for categorizing software (and hardware) vulnerabilities, which is also used in CVE reports and supported by many SAST tools. Each vulnerability in this enumeration has a unique identifier, accompanied by a description that explains how it relates to other vulnerabilities. This relationship is structured in the form of a *child-parent hierarchy*, where the child CWE represents a more specific vulnerability instance of the parent CWE. Top-level CWEs with no other parent CWEs are thus considered the lowest common denominator of all subordinate child CWEs. Accordingly, the top-level CWEs delineate broader *vulnerability classes*, while the subordinate ones denote specific *vulnerability types*.

Figure 3.3 on the following page provides an overview of the 20 vulnerability types[2] included in our SAST tool evaluation, which are grouped into the following vulnerability classes based on the CWE hierarchy:

Improper Control of a Resource Through Its Lifetime (CWE-664): This class refers to vulnerabilities caused by improper resource control. Examples include incorrect out-of-bounds read or write operations (CWE-{119, 125, 787}), use-after-free vulnerabilities (CWE-{415, 416}), resource management issues (CWE-{399, 770, 772, 401}), and incorrect type conversions (CWE-681).

Incorrect Calculation (CWE-682): This class refers to vulnerabilities caused by incorrect calculations whose results are then used in security-critical code. Examples include divide-by-zero (CWE-369) and integer-overflow vulnerabilities (CWE-190), which can lead to incorrect buffer-size calculations (CWE-131).

[2]Note that one included vulnerability specifies the vulnerability class CWE-664 as its type.

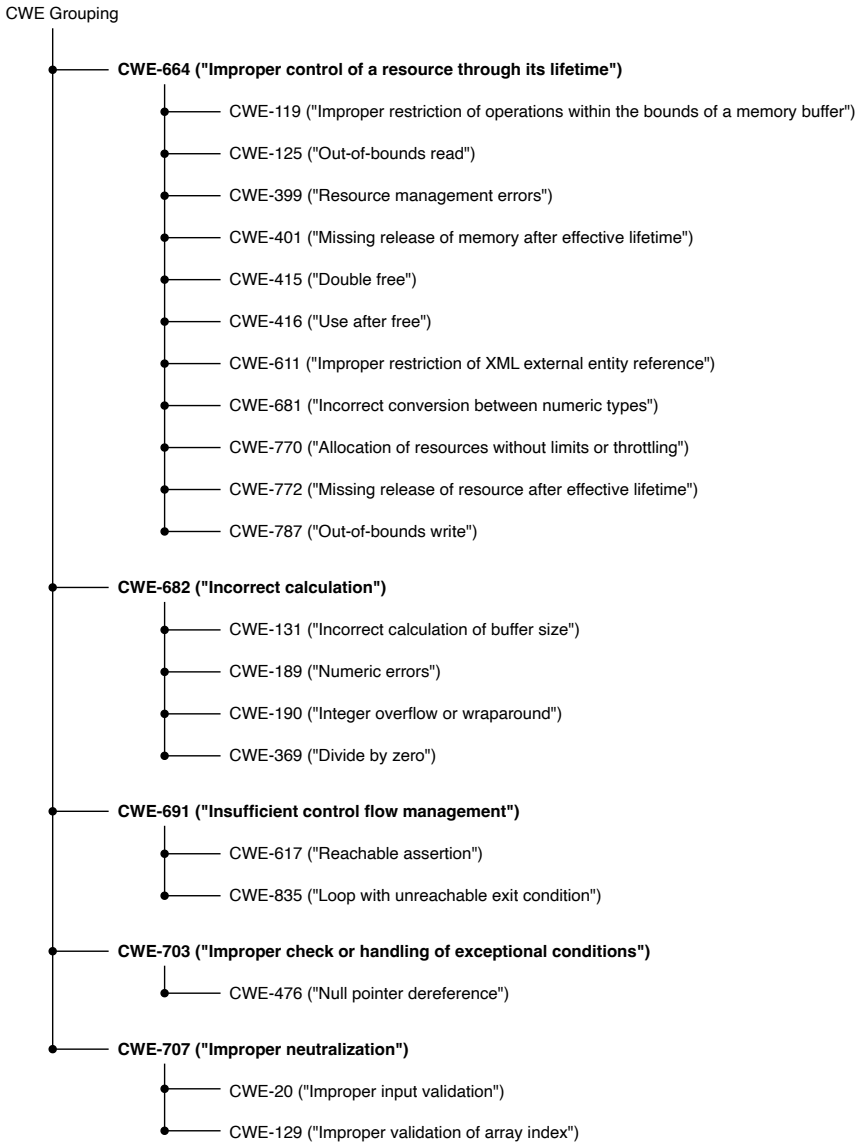


Figure 3.3.: Common Weakness Enumeration (CWE) grouping of the vulnerability types and classes included in our SAST tool evaluation.

Insufficient Control-Flow Management (CWE-691): This class refers to vulnerabilities caused by control-flow manipulations that compromise the program's security. Examples include loops whose exit condition is never reached or satisfied (CWE-835) and assertions that can be reached from the program entry (CWE-617), both allowing an attacker to trigger a DoS.

Improper Check or Handling of Exceptional Conditions (CWE-703): This class refers to vulnerabilities caused by improper inspection or handling of exceptional conditions in the code. Examples include missing if-statements that would prevent null-pointer dereferences (CWE-476), which often result in critical program crashes.

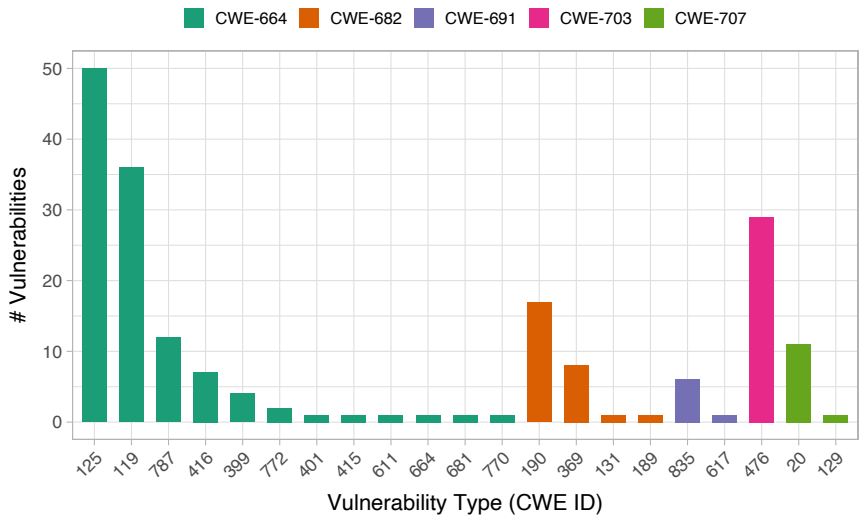


Figure 3.4.: Distribution of the vulnerability types (grouped by their vulnerability class) in our SAST tool evaluation (as appeared in [168]).

Table 3.2.: Supported vulnerability classes by each SAST tool (as appeared in [168]).

Vuln. Class	Flawfinder	Infer	Cppcheck	CodeChecker	CodeQL	CommSCA
CWE-664	✓	✓	✓	✓	✓	✓
CWE-682	✓	✓	✓	✓	✓	✓
CWE-703		✓	✓	✓	✓	✓
CWE-707	✓			✓	✓	✓
CWE-691	✓		✓		✓	✓

Improper Neutralization (CWE-707): This class refers to vulnerabilities caused by improper neutralization of program inputs against security risks. Examples include malformed strings passed via unvalidated program parameters or environment variables (CWE-20), as well as improper validation of array indices (CWE-129), both allowing attackers to execute (remote) code.

Figure 3.4 shows the distribution of the vulnerability types included in our evaluation. As mentioned earlier, each type belongs to one of the five vulnerability classes CWE-{664, 682, 691, 703, 707}, which contain 117, 27, 7, 29, and 12 vulnerabilities, respectively. These classes represent the most common and severe C/C++ vulnerabilities [106, 193, 328] and, as shown in Table 3.2, are supported (✓) by most of the selected SAST tools. A vulnerability class is thereby considered to be supported by an analyzer if its documentation indicates that a check is provided for at least one of the vulnerability types in that specific class. Notably, all static analyzers support the classes CWE-664 and CWE-682, which contain most of the vulnerabilities. However, Flawfinder does not support CWE-703, Infer and Cppcheck do not support CWE-707, and Infer and CodeChecker do not support CWE-691.

```
1  char *xmlMemStrdupLoc(const char *str, const char *file, int line) {
2      char *s;
3      size_t size = strlen(str) + 1;
4      MEMHDR *p;
5      if (!xmlMemInitialized) xmlInitMemory();
6      p = (MEMHDR *) malloc(RESERVE_SIZE + size);
7      if (!p) goto error;
8      p->mh_tag  = MEMTAG;
9      p->mh_size = size;
10     p->mh_type = STRDUP_TYPE;
11     p->mh_file = file;
12     p->mh_line = line;
13     xmlMutexLock(xmlMemMutex);
14     p->mh_number = ++block;
15     xmlMutexUnlock(xmlMemMutex);
16     // #define CLIENT_2_HDR(a) ((void *)(((char *)(a)) - RESERVE_SIZE))
17     s = (char *) HDR_2_CLIENT(p);
18     strcpy(s,str);
19     return(s);
20 error:
21     return(NULL);
22 }
```

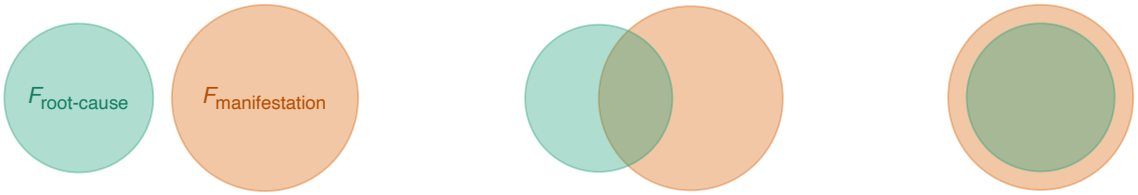
Listing 3.1.: Example of an out-of-bounds write vulnerability in Libxml2 (CVE-2017-5130) that shows the divergent code locations between its root cause in line 3 and the potential manifestations in lines 6, 18, and 19 flagged by FLAWFINDER, CODEQL and COMMSCA, respectively (as appeared in [168]).

Code Granularity for Vulnerability Detection

Given the extensive experimental setup we chose for our evaluation, we need to assess the effectiveness of the SAST tools in an automated manner. This requires defining criteria for approximating when a vulnerability is detected, specifically at what code granularity. In the following, we outline the process of determining an appropriate granularity for our evaluation.

SAST-Flagged Code Locations. In general, SAST tools identify and flag the code location(s) where a vulnerability potentially manifests as an erroneous program state rather than the location(s) of its root cause(s), which may be elsewhere in the code. This is because automated root cause analysis is complicated and, in some cases, nearly impossible [180]. Hence, to avoid wrongly flagged code locations (i.e., false positives), static analyzers tend to focus on the locations where the vulnerability might manifest.

Listing 3.1 shows a real-world example of a vulnerability in Libxml2 (CVE-2017-5130), where a potential integer overflow in `strlen` in line 3 (root cause) can manifest as an out-of-bounds write vulnerability (CWE-787) from line 6 onward. Interestingly, none of the executed analyzers identify the code with the root cause as vulnerable. Instead,



(a) RcMO = 0 **(b)** 0 < RcMO < 1 **(c)** RcMO = 1

Figure 3.5.: Visualization of the root cause-manifestation overlap (RcMO) value ranges for no, partial, and full overlap between a vulnerability's root-cause and manifestation function(s).

CodeQL and CommSCA flag line 6 as containing a CWE-401 ("missing release of memory after effective lifetime"). On line 18, CommSCA indicates a CWE-676 ("use of potentially dangerous function"), whereas Flawfinder and CodeQL report a CWE-120 ("buffer copy without checking size of input"). CodeQL also flags line 19 with a CWE-401, the same as in line 6. Hence, the SAST tools correctly flag the potential manifestations and suggest reasonable vulnerability types for the underlying vulnerability.

CVE-Provided Code Locations. Usually, the exact location(s) where the vulnerability manifests are not provided in the CVE reports [202], making it difficult to find an appropriate code granularity to approximate vulnerability detection. For example, some CVEs provide only the function name(s) where the vulnerability manifests, while others provide detailed descriptions of the root cause in the form of a software patch. These inconsistencies, coupled with our observation that SAST tools tend to flag the manifestation rather than the root cause location, made us choose the *function level* as the detection granularity. However, this requires that the root cause and the potential manifestation(s) of the vulnerabilities in our evaluation must occur within the same function(s). Otherwise, we would incorrectly count a vulnerability as missed when an analyzer flags a function other than the one provided by the CVE, regardless of whether it is a correct manifestation.

Definition 3.1 (Root Cause-Manifestation Overlap)**.** For the set of functions that contain the root cause ($F_{\text{root-cause}}$) of a vulnerability and the set where it manifests ($F_{\text{manifestation}}$), we define the proportional overlap of both sets as:

$$RcMO := \frac{\textbf{card}\left(F_{\text{root-cause}} \cap F_{\text{manifestation}}\right)}{\textbf{card}\left(F_{\text{manifestation}}\right)}$$

where **card**$(\cdot)$ returns the set cardinality.

Using the root cause-manifestation overlap (RcMO) metric from Definition 3.1 and visualized in Figure 3.5, Table 3.3 on the following page shows for each Magma subject program the percentage of contained vulnerabilities where (i) the root cause and manifestation location(s) are in disjoint functions (RcMO = 0), (ii) a partial function overlap exists (0 < RcMO < 1), and (iii) both locations fully overlap in the same function(s) (RcMO = 1)—the desired case. The results indicate that except for Libpng, where only

Table 3.3.: Percentage of the vulnerabilities in the different Magma subject programs with no, partial, and full overlap between their root-cause and manifestation function(s) (as appeared in [168]).

Subject	# Vulns.	Percentage of vulns. with RcMO = 0	0 < RcMO < 1	RcMO = 1
Libpng	7	43%	0%	57%
LibTIFF	13	0%	0%	100%
Libxml2	17	6%	0%	94%
OpenSSL	21	0%	0%	100%
PHP	15	7%	0%	93%
Poppler	22	0%	0%	100%
SQLite3	16	0%	0%	100%
Mean		8%	0%	92%

four out of the seven vulnerabilities (57%) overlap, in most other programs, the root cause and manifestation of the vulnerabilities occur within the same function(s). However, this analysis is only possible for the Magma programs, as Hazimeh et al. [114] provide additional information to those in the CVE reports about where the vulnerabilities may manifest in the code. For Binutils and FFmpeg, which are not part of Magma but are similar in program size, application domain, and vulnerability types they contain, we assume comparable RcMO results.

Summary (Detection Granularity)

In our evaluation, we compare the flagged and vulnerable code at the function level to assess the effectiveness of the SAST tools. We believe that this code granularity provides the most accurate and reliable approximation, as we have shown, that, on average, 92% of the Magma vulnerabilities have both the root cause (provided by the CVEs) and the corresponding manifestation(s) (flagged by the SAST tools) within the same function(s), assuming a similar pattern for Binutils and FFmpeg.

3.3. Evaluation Approach

In this section, we describe our automated approach to evaluating the effectiveness of SAST tools in detecting known vulnerabilities reported as CVEs. To analyze their effectiveness from different angles, we define different criteria for when a vulnerability is considered detected; some require only that the vulnerable code be flagged as problematic, while others additionally require the static analyzer to report a reasonable vulnerability type. In the following, we describe these detection criteria in more detail, as well as our approach to automatically verify that the vulnerability type reported by the analyzer conforms to that of the CVE.

		Comparison of vuln. class?	
		No	Yes
# Functions affected by vuln. to detect	≥ 1	Criterion 1	Criterion 2
	All	Criterion 3	Criterion 4

Figure 3.6.: Overview of the vulnerability detection criteria (as appeared in [168]).

3.3.1. Criteria for Vulnerability Detection

Inspired by Thung et al. [289], we evaluate the effectiveness of the SAST tools with respect to the following four *detection criteria*:

Criterion 1: A vulnerability is considered detected if at least one affected[3] function is flagged by the SAST tool, regardless of the reported vulnerability type.

Criterion 2: A vulnerability is considered detected if at least one affected function is flagged by the SAST tool together with a reasonable vulnerability type.

Criterion 3: A vulnerability is considered detected if all affected functions are flagged by the SAST tool, regardless of the reported vulnerability type.

Criterion 4: A vulnerability is considered detected if all affected functions are flagged by the SAST tool together with a reasonable vulnerability type.

These vulnerability detection criteria are also summarized in Figure 3.6. As the criterion number increases, the requirements for considering a vulnerability detected by a SAST tool become progressively stricter in terms of (i) the number of functions affected by the vulnerability that must be identified and (ii) whether the reported vulnerability type conforms to that of the underlying security bug. This also means that the more stringent the detection criterion, the less likely the risk of over-approximation due to random guesses by the analyzer that are misinterpreted as true positives when, in fact, they would be false positives if manually analyzed. Ultimately, analyzers that excel at these stricter criteria can significantly streamline the process of finding and mitigating vulnerabilities because they provide not only the vulnerable code location(s) but also an accurate, non-misleading vulnerability type.

3.3.2. CWE Mapping and Grouping

One challenge in automatically verifying that a SAST tool identifies the correct vulnerability type is that different analyzers use different identifiers for the vulnerabilities they support. For instance, FLAWFINDER uses the standardized CWE identifiers, while INFER uses its own vulnerability identifiers. To resolve this discrepancy, we map each analyzer-specific identifier to the corresponding analyzer-independent CWE identifier. An example of this

[3] By "affected", we mean that a function contains at least one faulty instruction that (along with others) causes the vulnerability.

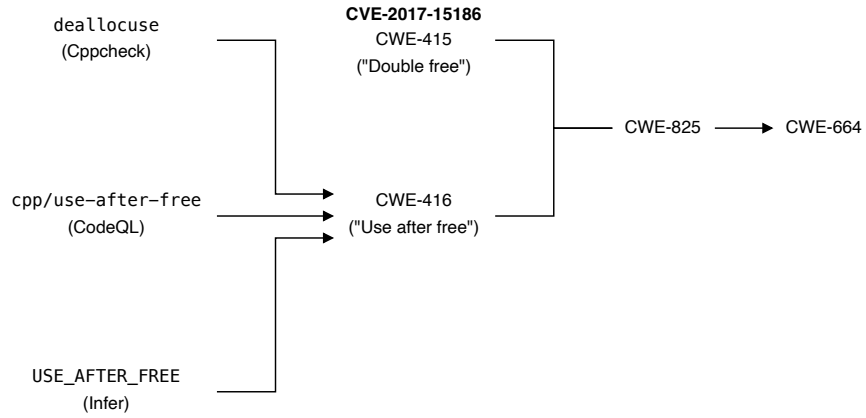


Figure 3.7.: Example of mapping the analyzer-specific use-after-free vulnerability identifiers of CPPCHECK, CODEQL, and INFER to the corresponding analyzer-independent vulnerability type CWE-416 (left part). The resulting CWE is then grouped together with the related CWE-415 into the vulnerability class CWE-664 (right part) (adapted from [168]).

mapping is shown in the left part of Figure 3.7, where the analyzer-specific use-after-free vulnerability identifiers of CPPCHECK, CODEQL and INFER are mapped to the corresponding analyzer-independent CWE-416 identifier.

However, directly comparing the low-level vulnerability types could lead to an unfair evaluation. In particular, if a static analyzer returns a closely related but not exact CWE identifier to the one specified in the CVE, the vulnerability would be considered missed in Criterion 2 and Criterion 4. As a result, the analyzer would be unfairly rated as less effective. To avoid this, we follow the approach of Goseva-Popstojanova and Perhinschi [102] and leverage the CWE hierarchy (described in Section 3.2.2 on page 31) in order to group related vulnerability types into more abstract classes, thus enabling a comparison at a higher CWE abstraction. An example of this grouping is shown in the right part of Figure 3.7, where the CWE-416 (“use after free”) SAST tool findings and the CWE-415 (“double free”) vulnerability (reported in CVE-2017-15186) are both grouped—via CWE-825 (“expired pointer dereference”)—into the CWE-664 (“improper control of a resource through its lifetime”) vulnerability class. Accordingly, we now correctly consider all three analyzers to have successfully detected the vulnerability in this example because the vulnerability types of the tool findings and that of the CVE all belong to the same overarching class.

3.4. Evaluation

In this section, we outline the setup for evaluating the effectiveness of the selected SAST tools in detecting vulnerabilities, describe the obtained evaluation results, and discuss the main findings and insights.

3.4.1. Setup

Research Questions

As mentioned before, this evaluation aims to verify whether state-of-the-art SAST tools can reliably identify potentially vulnerable code for our proposed fuzzing metric. Therefore, we answer the following *research questions* through empirical evaluations:

RQ 1a: SAST Tool Effectiveness. How effective are modern SAST tools at detecting vulnerabilities in real-world C/C++ programs?

RQ 1b: SAST Tool Combination Tradeoff. What is the tradeoff between the number of vulnerabilities detected and the number of functions flagged when using the best SAST tool combination instead of the best single analyzer?

RQ 1c: Best and Worst SAST-Detected Vulnerabilities. Which vulnerability classes (included in our evaluation) are detected best and worst by SAST tools?

In the first research question, we evaluate the effectiveness of each analyzer individually to understand how the different SAST techniques perform in detecting vulnerabilities. Then, in the second question, we examine the overall effectiveness of SAST, i.e., when using multiple analyzers in combination, which helps decide if SAST tools can be employed for our vulnerability-oriented code coverage metric. Finally, we investigate the strengths and weaknesses of SAST in detecting vulnerabilities within the different vulnerability classes included in our evaluation. In particular, this analysis reveals whether potential limitations affect all or only certain vulnerability classes, thus facilitating the identification of countermeasures to improve SAST effectiveness.

Metrics

We use the true positive rate (TPR) and flagged code rate (FCR) metrics defined in Equations (2.1) and (2.2) on page 14 with respect to the four detection criteria (see Section 3.3.1 on page 39) to evaluate the SAST tool effectiveness.

Infrastructure

We performed all experiments on a machine with an Intel® Xeon® E5-1650v2 central processing unit (CPU) containing 12 logical cores that run at 3.5 GHz, with access to 128 GB random-access memory (RAM) and GNU/Linux Ubuntu 16.04 (64-bit) as the operating system.

3.4.2. Results

Note that CodeQL did not produce any analysis results for the Binutils programs despite several days of running and multiple retries. For this reason, we rate CodeQL as having found zero vulnerabilities in Binutils.

Figure 3.8.: Percentage of vulnerabilities detected by each SAST tool in each subject program (as appeared in [168]).

SAST Tool Effectiveness

Figure 3.8 shows the percentage of vulnerabilities detected by each SAST tool and in each subject program. The results indicate that many vulnerabilities were not found, especially in the subjects Poppler, FFmpeg, and Libpng. This may be due to the following reasons. Regarding Poppler, it is the only C++ program in our evaluation. Although the selected analyzers claim to support C++, they seem to focus primarily on plain C, with only rudimentary support for C++. Regarding FFmpeg, the largest subject program in our evaluation with 413,353 LoC, its size may force the SAST tools to reduce their analysis precision once the code complexity, such as the depth of nested `if`- or `#ifdef`-statements, reaches a certain limit. Regarding Libpng, the low detection rates may be attributed to the discrepancies between the functions that contain the root cause and those where the vulnerabilities manifest (see Table 3.3 on page 38). Despite these program-specific issues, the overall results are consistent between subjects with front-ported vulnerabilities (i.e., the Magma programs) and those with non-front-ported vulnerabilities (i.e., the Binutils programs and FFmpeg).

Figure 3.9 on the next page provides an aggregated view of the previous results, showing the percentage of vulnerabilities detected versus functions flagged (mirrored on the y-axis) by each SAST tool across all subject programs. The results indicate significant per-

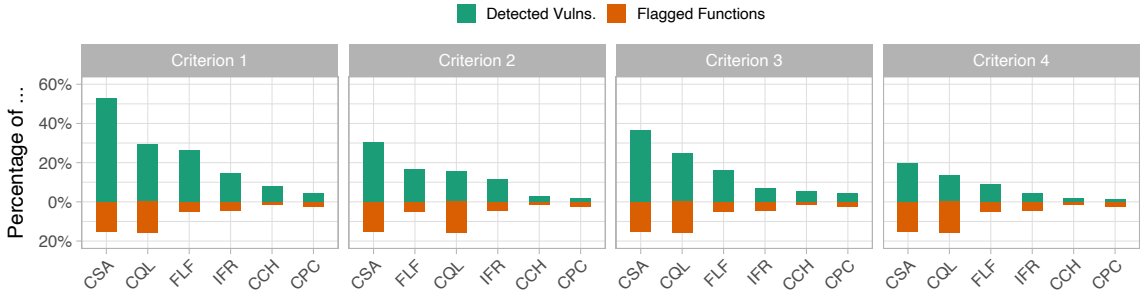


Figure 3.9.: Percentage of vulnerabilities detected versus functions flagged (mirrored on the y-axis) by each SAST tool across all subject programs (as appeared in [168]).

Figure 3.10.: Percentage of vulnerabilities detected versus functions flagged (mirrored on the y-axis) by the best free and commercial single SAST tools and analyzer combinations across all subject programs (as appeared in [168]). The commercial tool CommSCA is highlighted in bold.

formance differences between the analyzers, with CommSCA, CodeQL, and Flawfinder being the most effective ones. In particular, the commercial tool CommSCA outperforms the best free analyzers, CodeQL and Flawfinder, by detecting 45 (23.4pp), 26 (13.5pp), 22 (11.5pp), and 12 (6.3pp) more vulnerabilities across the four criteria, respectively. Furthermore, CommSCA flags slightly fewer functions than CodeQL, making it less prone to false positives. Another observation is that Flawfinder finds more vulnerabilities than Infer across all detection criteria with a similar number of flagged functions. Moreover, Flawfinder achieves roughly the same detection rates as CodeQL for the first two criteria while flagging 11pp fewer functions.

Summary (RQ 1a)

Our evaluation shows that state-of-the-art SAST tools can detect real-world vulnerabilities in C/C++ programs; however, they also miss a significant number. In particular, the best-performing analyzer (CommSCA) detects 101 (52.6%) of the 192 real-world vulnerabilities under Criterion 1, 58 (30.2%) under Criterion 2, 70 (36.5%) under Criterion 3, and only 38 (19.8%) under Criterion 4.

SAST Tool Combination Tradeoff

To evaluate the performance increase when using multiple analyzers, we select the free and commercial SAST tools, as well as the combinations thereof, that identify the most vulnerabilities. A vulnerability is then considered detected if at least one of the analyzers in the combination was able to identify it under the respective detection criterion. Since multiple analyzer combinations detect the same number of vulnerabilities, we choose those that involve the fewest tools, as they are likely to produce the fewest false positives.

Figure 3.10 on the preceding page shows the results of this analysis, i.e., the percentage of vulnerabilities detected versus functions flagged (mirrored on the y-axis) by the best free and commercial single SAST tools and analyzer combinations across all subject programs. Interestingly, the combinations with CommSCA consist of less than six analyzers, indicating that CommSCA covers all the vulnerabilities detected by Cppcheck under Criterion 1 and Criterion 3, and by CodeChecker under Criterion 2 and Criterion 4. Also, most of the combinations shown include Infer, Cppcheck, and CodeChecker, which are less effective when used individually (see Figure 3.9 on the previous page). This suggests that they manage to find vulnerabilities that the other tools miss.

These results also show that the effectiveness of SAST can be significantly increased by combining analyzers, supporting the recommendation of Fatima et al. [82] to use multiple tools whenever possible. In particular, the best-performing combination (Flawfinder-Infer-CodeQL-CodeChecker-CommSCA) detects 34 (17.7pp) more vulnerabilities under Criterion 1 and 39 (20.3pp) more under Criterion 3 than the best single analyzer (CommSCA), while flagging 14.6pp more functions. Under Criterion 2 and Criterion 4, the same combination outperforms CommSCA by 24 (12.5pp) and 21 (10.9pp) more vulnerabilities detected, respectively, also with 14.6pp more functions flagged. Notably, under Criterion 1, Criterion 3, and Criterion 4, it detects more than twice as many vulnerabilities as CodeQL, albeit at the cost of flagging about twice as many functions. Under Criterion 2, it detects three times as many vulnerabilities as Flawfinder, with six times more functions flagged. Also, note that the best combinations of free static analyzers detect at least as many vulnerabilities as the commercial tool CommSCA across all four criteria.

Summary (RQ 1b)

Our evaluation shows that the best-performing SAST tool combination can increase the detection rate by 10.9 to 20.3 percentage points (pp) more vulnerabilities found than the best single analyzer while flagging 14.6pp more functions. In particular, it detects 135 (70.3%) of the 192 vulnerabilities under Criterion 1, 82 (42.7%) under Criterion 2, 109 (56.8%) under Criterion 3, and 59 (30.7%) under Criterion 4, indicating that some vulnerabilities remain undetected.

Best and Worst SAST-Detected Vulnerabilities

Finally, to better understand the strengths and weaknesses of SAST, we analyze which vulnerability classes are detected best and worst. For this, Figure 3.11 on the facing page shows the proportion of vulnerabilities detected in each class when all six SAST tools are

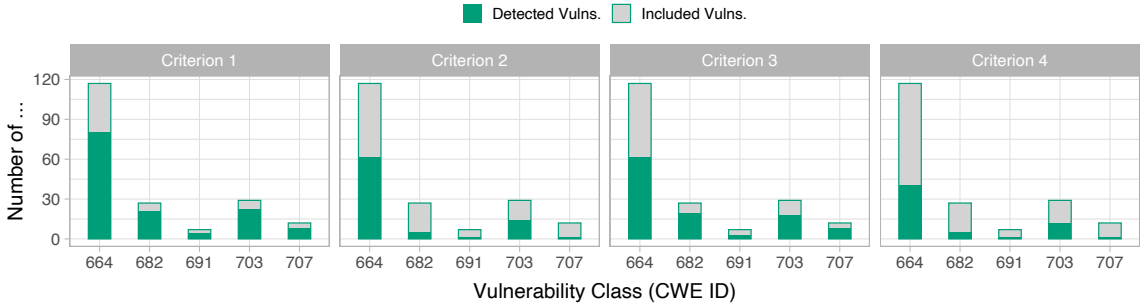


Figure 3.11.: Proportion of vulnerabilities detected in each vulnerability class when using all six SAST tools (as appeared in [168]).

used in combination. Note that Criterion 1 and Criterion 3 ignore the vulnerability type suggested by the analyzers and thus provide only limited insights here.

The two vulnerability classes that are supported by most of the static analyzers (see Table 3.2 on page 35), CWE-664 ("improper control of a resource through its lifetime") and CWE-703 ("improper check or handling of exceptional conditions"), are also the ones whose vulnerabilities are most effectively detected under Criterion 2 and Criterion 4. However, roughly less than half of the included vulnerabilities in these two classes could be detected under these two criteria, highlighting the limited effectiveness of current SAST techniques. Moreover, vulnerabilities categorized as CWE-682 ("incorrect calculation"), which are supported by all six static analyzers, are primarily detected under Criterion 1 (77.8%) and Criterion 3 (70.4%), suggesting that the respective tools do not accurately distinguish between vulnerability types.

The least effectively detected vulnerability classes, CWE-691 ("insufficient control-flow management") and CWE-707 ("improper neutralization"), are also the least supported ones. Here, only one in seven (14.3%) of the CWE-691 vulnerabilities and one in twelve (8.3%) of the CWE-707 vulnerabilities could be detected under Criterion 2 and Criterion 4. Although Cppcheck supports CWE-691, CodeChecker CWE-707, and Flawfinder, CodeQL, and CommSCA both classes, the likelihood of finding such vulnerabilities via SAST is very low.

Summary (RQ 1c)

Our evaluation shows that CWE-{664, 703} vulnerabilities are more effectively detected via SAST than CWE-{682, 707, 691} ones. In particular, 40 to 80 (34.2%–68.4%) of the 117 CWE-664 vulnerabilities, 12 to 22 (41.4%–75.9%) of the 29 CWE-703 ones, 5 to 21 (18.5%–77.8%) of the 27 CWE-682 ones, 1 to 4 (14.3%–57.1%) of the 7 CWE-691 ones, and 1 to 8 (8.3%–66.7%) of the 12 CWE-707 ones across the different detection criteria. This indicates that even many of the best-supported vulnerabilities were missed.

3.4.3. Discussion

The commercial analyzer CommSCA has emerged as the most effective SAST tool in our evaluation. Also, CodeQL stands out as the best free analyzer, which has a lower detection

rate than CommSCA while flagging a similar number of functions. However, CodeQL might have surpassed CommSCA if it had returned any results for Binutils; unfortunately, CodeQL's analysis did not terminate even after several days of running and multiple retries. Nonetheless, the results of our evaluation suggest that the performance of free and commercial analyzers is comparable. Among the freely available tools, CodeQL is closely followed by Flawfinder, which even outperforms CodeQL in some cases, thereby flagging only about a third of the functions compared to CodeQL. Notably, Flawfinder consistently outperforms the more sophisticated analyzers Infer, CodeChecker, and Cppcheck, suggesting that semantic SAST techniques do not always outperform the simpler syntactic techniques. This is partly because C/C++ memory-safety vulnerabilities often manifest in memory (de)allocation or string utility functions (of the C standard library) and are, therefore, relatively easy to find via pattern matching. However, many vulnerabilities were not found when the analyzers were run individually, contrasting the high detection rates in related studies that use artificial bugs. Hence, using a single SAST tool is very unreliable in identifying bug-prone code.

Analyzing the combined effectiveness of multiple SAST tools, we find that almost all vulnerabilities detected by Cppcheck and CodeChecker were also detected by CommSCA, suggesting that CommSCA can be used instead to reduce the execution costs. Furthermore, the most effective analyzer combinations include Infer, which is less effective when run alone but detects vulnerabilities that the other tools also support but overlook. Despite the drawback of flagging more functions (and thus likely more false positives), using multiple analyzers not only covers more different vulnerability types but also increases the chances of detecting them. However, this increase is still insufficient for our vulnerability-oriented fuzzing metric, as about 30% of the vulnerable functions are missed under the least restrictive detection criterion. This limitation thus prevents a reliable assessment of fuzzing effectiveness, as a campaign that covers potentially vulnerable code that is not flagged as such by the analyzers would be incorrectly rated as less effective.

The missed vulnerabilities thereby affect not just one but all of the included vulnerability classes, with those involving memory-safety vulnerabilities being slightly more effectively detected than others. This consistency with which the vulnerabilities are missed suggests that certain vulnerabilities—probably due to the code abstractions on which the respective SAST techniques operate—are inherently difficult to detect via SAST. In particular, their analysis may be too coarse-grained or may focus solely on individual functions without considering the calling functions, thereby missing crucial information needed to identify subtle or interrelated vulnerabilities. This problem is exacerbated in large software systems, where the code complexity may force the static analyzer to further reduce its analysis accuracy.

3.5. Threats to Validity

Achieving generalizable results in empirical software engineering or testing research is challenging, if not impossible, due to the context-specific nature of software. Therefore, the results and conclusions drawn from our evaluation may vary for SAST tools and subject programs other than those used in our study. Nonetheless, we conducted a large-

scale empirical evaluation to ensure a certain degree of *external validity* and, further, to gain some insights into the reliability of SAST tools in identifying potentially vulnerable code. Specifically, we analyze the effectiveness of six different static analyzers (including one commercial tool) that implement advanced SAST techniques on several real-world programs that contain many different vulnerabilities.

The threats discussed hereafter primarily concern the *internal validity*, i.e., potential methodological mistakes in our study. The first threat relates to the correctness of the CWE mapping that we created for Infer and CodeChecker since both analyzers do not support CWEs by default. To reduce the risk of incorrect mapping, we carefully reviewed the description of each analyzer-specific vulnerability identifier to ensure that we assign an appropriate analyzer-independent CWE identifier. For identifiers without a clear description, we contacted the developers for additional information or asked them directly to validate our mapping.

Furthermore, whenever real-world programs are used in such benchmarks, there is a possibility that they may contain unknown vulnerabilities. Due to our automated evaluation approach, we did not consider new vulnerabilities discovered by the static analyzer. Yet, we believe that examining the analyzers' ability to identify known vulnerabilities allows drawing valid conclusions about their effectiveness, albeit mostly with respect to more complex vulnerabilities in mature programs. That said, the SAST tools may perform better on newer, less thoroughly tested source code. However, since our proposed fuzzing metric should be applicable to all kinds of software, we consider it appropriate to evaluate the analyzers on more challenging programs.

Another potential threat arises from the front-ported Magma vulnerabilities. In particular, the integration of older vulnerable code into newer program versions may adversely affect the static bug finding. To mitigate this potential bias, we removed all "canary" code[4] in the Magma programs to ensure that only the vulnerable code is analyzed. In addition, we include 82 non-front-ported vulnerabilities from programs outside of Magma, for which we did not observe any differences in terms of SAST tool effectiveness.

A related threat involves our assumption that the code locations containing the root cause (as provided by the CVEs; ground truth) and those representing the potential manifestation(s) (as flagged by the analyzers) are generally within the same functions. In fact, there are two parts to this threat. First, the Magma vulnerabilities that we use for the root cause-manifestation overlap analysis may introduce a selection bias because they are particularly easy to detect due to this high overlap of root cause and manifestation location(s)—a threat we cannot exclude. Second, this high overlap may not be accurate for the Binutils and FFmpeg vulnerabilities, for which we could not perform the same analysis due to missing data. However, since these programs are similar to the Magma programs in terms of program size, application domain, and vulnerability types contained, we believe that an equally high overlap is reasonable to assume.

The final threat concerns the question of whether SAST tools have already been used on our subject programs, which could affect the conclusions drawn from the results, both positively and negatively. On the positive side for SAST, vulnerabilities may have been

[4]Such canaries are code-level sanitizers that enforce a fuzzer-detectable program crash upon encountering a security violation at run time [114].

effectively detected by static analyzers during development and never officially reported as CVEs. The remaining vulnerabilities are then generally difficult to identify via SAST. On the negative side, such analyzers may have originally found the included vulnerabilities, yet newer tools do not detect them anymore. Although we have not found any obvious evidence of SAST tools being used in the respective repositories, we cannot completely rule out this possibility.

3.6. Summary

In this chapter, we evaluated the effectiveness of several state-of-the-art SAST tools in detecting known real-world software vulnerabilities to see if they can reliably identify potentially vulnerable code for our vulnerability-oriented code coverage metric. Our evaluation shows that while such static analyzers can detect more complex vulnerabilities, they also miss a significant number. Although the effectiveness of static bug-finding can be improved by using multiple analyzers in combination (at the cost of more flagged code), this approach still misses many vulnerabilities across many different vulnerability classes. This suggests that certain vulnerabilities are inherently difficult to detect with SAST, precluding its use in our proposed fuzzing metric. Therefore, in the next chapter, we explore approaches for integrating SAST with other vulnerability prediction techniques to identify potentially vulnerable code more effectively.

4. ML-Based Vulnerability Prediction

This chapter researches the integration of code complexity metrics and SAST tool findings into a vulnerability prediction model using machine learning to identify potentially vulnerable functions more reliably and accurately than using SAST tools alone. Parts of this chapter have appeared in a peer-reviewed publication [170], first-authored by the author of this thesis.

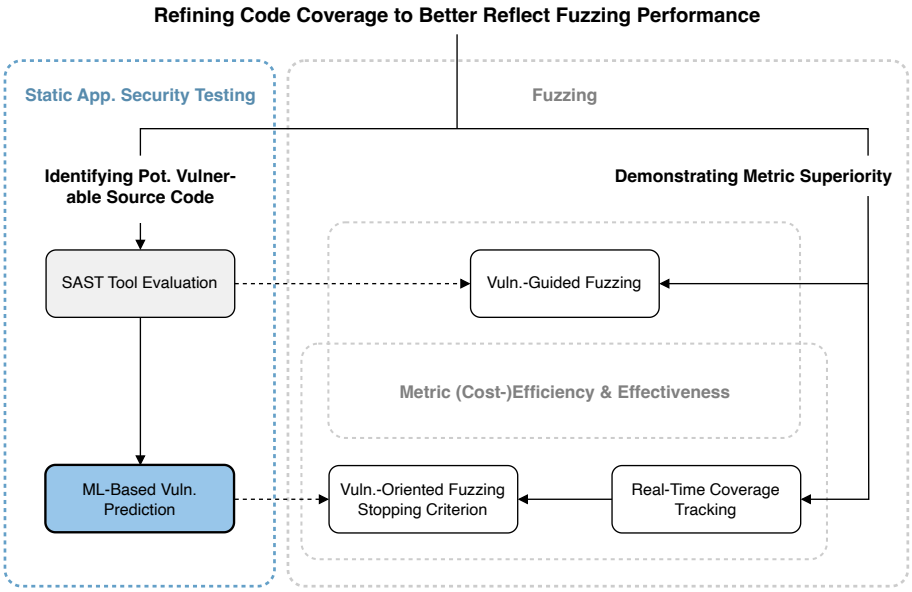


Figure 4.1.: Big Picture (Contribution 2: ML-Based Vulnerability Prediction).

4.1. Introduction

Context. As shown in Figure 4.1, this chapter explores approaches to statically identify potentially vulnerable code beyond the conventional use of SAST tools. The previous chapter showed that while such analyzers can detect real-world vulnerabilities, they also miss many (across different vulnerability classes), even when using multiple advanced tools in combination. This suggests that certain vulnerabilities are *inherently* difficult to detect via SAST, probably due to the code abstractions used to scan the source code for security bugs in a reasonable time. As a result, subtle or interconnected vulnerabilities may generally fall outside the scope of SAST, making it difficult, if not impossible, to improve the analyzers' effectiveness.

State-of-Practice and Problem. Experts recommend adapting static analyzers to the targeted codebase by fine-tuning their checks [83, 221, 290, 320], as well as complementing SAST with secure coding [220] or code reviews [16, 29, 45, 111]. While such optimizations can be helpful, the large number of missed vulnerabilities that we observed in our evaluation points to fundamental limitations of SAST, given that we have already enabled all security-related SAST checks supported by the selected tools. Therefore, we do not believe that further configuration tweaks or enhancements to these checks would improve their effectiveness sufficiently to reliably apply our vulnerability-oriented fuzzing metric. Furthermore, optimizing the checks for the targeted codebase requires in-depth domain knowledge and manual effort, limiting the practicality of our metric.

Besides SAST, various *vulnerability heuristics* based on code complexity [73, 187, 268, 337], code change rate [207, 268, 336], or bug pattern mining [140, 297, 317] have been proposed to automatically identify bug-prone code. While these heuristics have proven effective in the cited papers, they tend to focus on specific types of bugs rather than on multiple types. An approach that has yet to be explored is the integration of SAST with such vulnerability heuristics to address the limited effectiveness of SAST. In particular, we hypothesize that combining the two approaches will increase the reliability with which potentially vulnerable code is detected.

Solution Approach. To address this research gap, we integrate vulnerability indicators derived from SAST tool findings and established code complexity measures into a vulnerability prediction model using ML [71]. Given the varying effectiveness of the various indicators [169, 268] that we plan to combine, we choose to use ML to find and determine the appropriate *impact* of each indicator on the model's final prediction. Specifically, our ML-based approach automatically learns the effectiveness of each indicator from codebases with known vulnerabilities and then assigns appropriate weights for their prediction. Hence, indicators that have proven effective in identifying known vulnerabilities are assigned higher weights and thus have a greater impact on the overall prediction. We believe that assigning weights based on the indicators' effectiveness in finding known vulnerabilities is promising because vulnerabilities similar to those found in the past are often reintroduced through code regression [281, 282].

Contributions. Overall, we contribute a qualitative analysis of 25 vulnerability indicators derived from SAST tool findings and code complexity metrics, which we combine into different ML vulnerability prediction models to reliably identify bug-prone functions. Our evaluation shows that most of the trained models effectively discriminate between vulnerable and (likely) non-vulnerable functions in real-world programs with known security bugs, achieving true and false positive rates above 0.9 and below 0.5, respectively.

4.2. Vulnerability Prediction Approach

In this section, we present our ML-based approach for predicting potentially vulnerable code. Specifically, we describe the feature selection and engineering, as well as the training process and application of the final prediction model.

Table 4.1.: Function-level machine learning features employed for vulnerability prediction. Entries postfixed with `[_LoC]` denote two features: one with the absolute value (without the postfix) and one with the feature value relative to the lines of code (LoC) of the corresponding function (with the postfix).

Category	Identifier	Description
Complexity	`LoC`	Lines of code
	`CCN[_LoC]`	Cyclomatic complexity number
	`N_In|Out_Calls[_LoC]`	Number of incoming/outgoing function calls
Tools	`N_FLF|CPC|IFR|CQL|CCH _Flags[_LoC]`	Number of lines flagged by Flawfinder, Cppcheck, Infer, CodeQL, and CodeChecker, respectively
	`N_SAST_Flags[_LoC]`	Number of lines flagged by all SAST tools
	`N_ASAN_Flags[_LoC]`	Number of lines flagged by AddressSanitizer
	`N_SAST+ASan_Flags[_LoC]`	Number of lines flagged by all SAST tools plus ASan
	`N_SAST_Tools`	Number of different SAST tools that flagged a function
	`N_SAST+ASan_Tools`	Number of different tools that flagged a function

4.2.1. Feature Engineering

In ML, a feature refers to a single measurable property (i.e., piece of information) used by a model to predict a phenomenon [104]. Consequently, the prediction accuracy is directly influenced by the feature's ability, and thus its quality, to capture and reflect the characteristics of the phenomenon being predicted. Below, we outline the features selected for our ML vulnerability prediction models, including the methods we use to filter out irrelevant features that may degrade the prediction performance.

Feature Selection

Our goal is to compile a set of lightweight features that can be easily and directly extracted from the SUT's source code or from readily available test artifacts (e.g., reports generated by SAST tools) yet are indicative of potentially vulnerable code. Table 4.1 lists the 25 function-level ML features we employ for vulnerability prediction, which can be categorized into *complexity-based* and *tool-based* features. We thereby take the absolute values of the features and, where possible, the values relative to the function LoC. This normalization ensures that the feature values are comparable across functions of different sizes, preventing large functions from disproportionately influencing the model training and, further, the prediction simply by containing more lines.

Complexity-Based Features. The features in this category are based on the intuition that the more complex a function is, the more difficult it is to understand and, therefore, the more likely it is to introduce vulnerabilities. We distinguish between *intra-function* and *inter-function* complexity.

Intra-function complexity refers to the internal code complexity of a function, for which we use two measures, lines of code and cyclomatic complexity number (CCN) [183]. The latter is the number of possible linearly independent paths through the function's CFG. We thereby chose these two measures because they have proven effective as vulnerability indicators in previous research [73, 187, 268, 269].

Inter-function complexity refers to the external interaction complexity of a function. Here, we chose centrality measures such as the number of incoming and outgoing function calls, which can be computed by counting the corresponding edges in the SUT's call graph (CG). Both measures have been successfully applied to vulnerability prediction [337] and fault localization [330].

Tool-Based Features. The features in this category are derived from the findings of the SAST tools described in Section 3.2.1 on page 30, as well as the (memory) sanitizer[1] AddressSanitizer (ASan) [263]. Similar to Österlund et al. [222], we consider all code regions (here functions) as potentially vulnerable that contain at least one sanitizer-inserted check. Since sanitizers insert many checks into the source code, they miss fewer vulnerable functions than SAST tools. We thereby chose AddressSanitizer instead of another sanitizer [147] because it supports the most common C/C++ vulnerabilities, such as buffer overflows, memory leaks, and freed memory usage.

The tool-based features we selected are based on two assumptions. First, the more lines in a function that are flagged by the tools, the more likely it is that the function is vulnerable. Second, the more different tools agree that a function is bug-prone, the more likely it is to be vulnerable. The features based on the first assumption have already been successfully used as vulnerability indicators by Pereira et al. [230], while we propose those that are based on the second assumption.

Feature Filtering

To avoid overfitting the ML vulnerability prediction models to the training set, i.e., the feature values extracted from the training programs in Table 4.2a on page 58, we eliminate all *uninformative* and *redundant* features [44, 104].

Regarding the former, we remove all features with (near-)zero variance because they hardly discriminate between potentially vulnerable and non-vulnerable functions. Here, we found that the function LoC (feature `LoC`) and the absolute and relative number of lines per function flagged by Cppcheck, Infer, and CodeChecker (`N_CPC|IFR|CCH|_Flags` and `N_CPC|IFR|CCH_Flags_LoC`) vary little across our training set, so we omit them.

Regarding the latter, we identify pairs of highly correlated features and remove one from each pair, as both features provide similar insights. Specifically, we target feature pairs with a Pearson [259] correlation score of 0.8 and higher, which, as shown in Figure 4.2 on the facing page, exists between the features `N_ASan_Flags` and `N_SAST+ASan_Flags` as well as between `N_ASan_Flags_LoC` and `N_SAST+ASan_Flags_LoC`.

[1] Such error detectors instrument the SUT at compile time with additional (security) checks that abort the program execution once an unsafe program state is encountered at runtime [274].

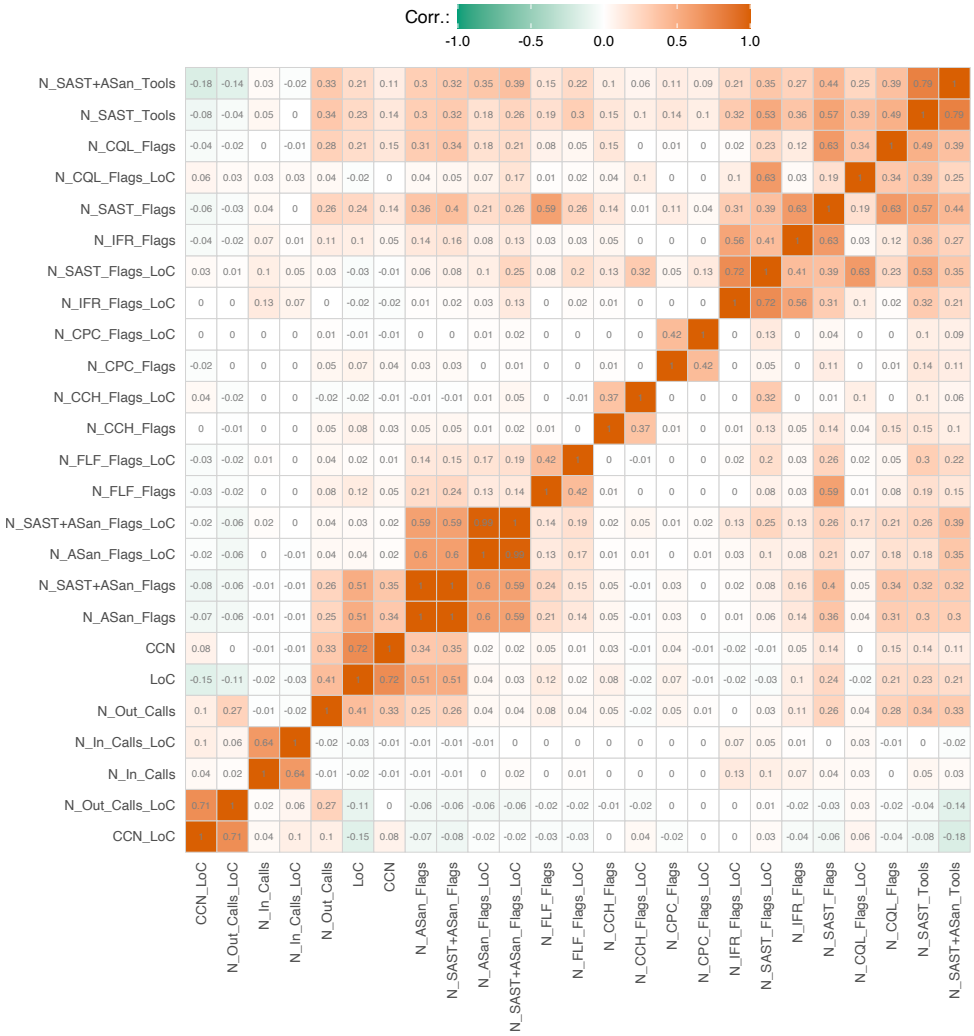


Figure 4.2.: Correlation scores between the selected ML vulnerability prediction features, as measured by the Pearson correlation coefficient. Scores highlighted in green indicate a negative correlation, while those in orange indicate a positive correlation.

4.2.2. Machine Learning Models

There are several algorithms for training ML classification models, each offering distinct advantages. Below, we outline the most commonly used algorithms/models for vulnerability prediction [131, 225], with the most effective one(s) then used to find potentially vulnerable code for our vulnerability-oriented code coverage metric.

Generalized Linear Model (GLM): This model extends *linear regression*[2] by establishing a relationship between the linear combination of features plus their associated weights and the probabilities of the prediction classes using a link function. For

[2]The objective of linear regression is to find a straight line in two dimensions or a (hyper)plane in a higher-dimensional space that best fits the training data by minimizing the distances between the data points and the line or plane [182].

this, the training algorithm begins with default feature weights, which are then iteratively adjusted to minimize the prediction error (i.e., the difference between the predicted and actual outcome). For making predictions using the trained model, the feature values of new, unseen data are multiplied by the learned weights, with the resulting linear combination then passed through the link function to determine the class probabilities and, thus, the final classification. Thereby, GLMs support the analysis of each feature's contribution to the final prediction (i.e., feature importance). However, they are sensitive to outliers, as well as to over- and underfitting [210].

Feedforward Neural Network (FNN): This model consists of several connected nodes, which are grouped into layers. The first layer is the *input layer*, where each node represents a single feature. It is followed by one or more *hidden layers* that compute weighted sums of the input feature values, thereby applying a (non-linear) *activation function* to regulate the node's impact on the prediction. Finally, the *output layer* transforms the internal data into a class label. To train such a model, a *backpropagation* algorithm iteratively adjusts the weights of the node connections to minimize the prediction error. For making predictions, the new data point is fed into the input layer and propagated through the network's connections, layer by layer. Each layer applies the learned weights and the activation function until the data reaches the output layer, producing the class probability scores and the final classification. This approach allows FNNs to learn complex non-linear data relationships, making them suitable for various prediction tasks. However, they are sensitive to the assigned weights, which can get stuck in local minima [252].

Random Decision Forest (RDF): This model integrates multiple independent *decision trees*[3], each acting as a branching questionnaire, determining the final classification through *majority voting*. For this, the training algorithm splits the dataset into several subsets using random sampling (with replacement). A separate decision tree is then trained on each subset using a random subset of features to ensure diversity. This is done by recursively splitting the data subset to create nodes that optimally separate the output classes based on the respective feature values until it reaches a precise prediction for each branch. For making predictions, each tree in the random forest classifies the new data point based on its subset of features. The final prediction is then determined by the class that has received the most votes among the trees. Due to the use of multiple decision trees, RDFs are robust to noisy data and outliers. Furthermore, they support analyzing the importance of the features in the prediction. However, such decision forests can be very memory intensive, especially when using many trees [30].

Gradient Boosted Machine (GBM): Like random forests, this model combines several weak learners, such as decision trees, into a more robust ensemble. However, they differ in that the weak learners depend on each other, with each subsequent learner focusing on *correcting* the prediction errors of the previous one (boosting mechanism).

[3] A decision tree is a predictive model structured in a tree-like manner. Therein, each internal node represents a decision rule and each leaf node a prediction outcome [244].

The training algorithm starts with a basic model that makes initial predictions on the training set. It then analyzes the prediction errors and trains the next weak learner to correct them. This process is repeated, with each new learner addressing the residual errors of the previous one(s), thereby improving the overall prediction accuracy. For making predictions, the new data point flows sequentially through the model, with each sub-model refining the prediction. The final output is then an aggregate of the sub-models' contributions. Accordingly, GBMs share many of the advantages and disadvantages of regular RDFs. However, they often yield better predictions because of this boosting mechanism [88].

Support Vector Machine (SVM): Similar to GLMs, this model uses *linear boundaries* to perform classification tasks. For this, it projects data points onto a higher-dimensional feature space to find the optimal (hyper)plane separating the output classes. However, unlike GLMs, the training algorithm does not model probabilities associated with these classes. Instead, it directly searches for the hyperplane that maximizes the margin, i.e., the distance between the hyperplane and the nearest data points (*support vectors*) from each class. A larger margin indicates better class separation, ultimately leading to a more robust prediction. SVMs are thereby particularly effective in high-dimensional spaces where the outcomes have a clear margin of separation. Accordingly, such models are frequently used for binary classification, predicting the class of a new data point based on the side of the hyperplane on which it falls. However, they are sensitive to noisy data and struggle with overlapping output classes [60].

4.2.3. Model Training

Based on the feature values extracted from codebases with known vulnerabilities, we train several binary ML classification models to predict whether a function is potentially vulnerable or not. We build and evaluate several models using the ML training algorithms described above to see which ones work best for our purposes. Below, we describe the training configuration used to build the different models, followed by the importance of the features in their vulnerability predictions.

Training Control

Before training, we *normalize* each feature value between 0 and 1 using a *min-max scaler* to avoid bias in training algorithms that use distance calculations.

Also, since our training set is unbalanced, containing 121 vulnerable functions versus 43,963 non-vulnerable functions (see Table 4.2a on page 58), we *downsample*[4] the feature data of the majority output class so that both classes are equally often represented.

Furthermore, we follow the best practices suggested by Arp et al. [9] and train and evaluate our ML vulnerability prediction models using *10-fold cross-validation*[5] over all

[4] When upsampling the training set, we obtained almost identical evaluation results.

[5] k-fold cross-validation divides the dataset into k equal folds, training the model on $k-1$ folds and testing it on the remaining one. This process is repeated k times, each time using a different fold as the testing set, to ensure a robust model evaluation [311].

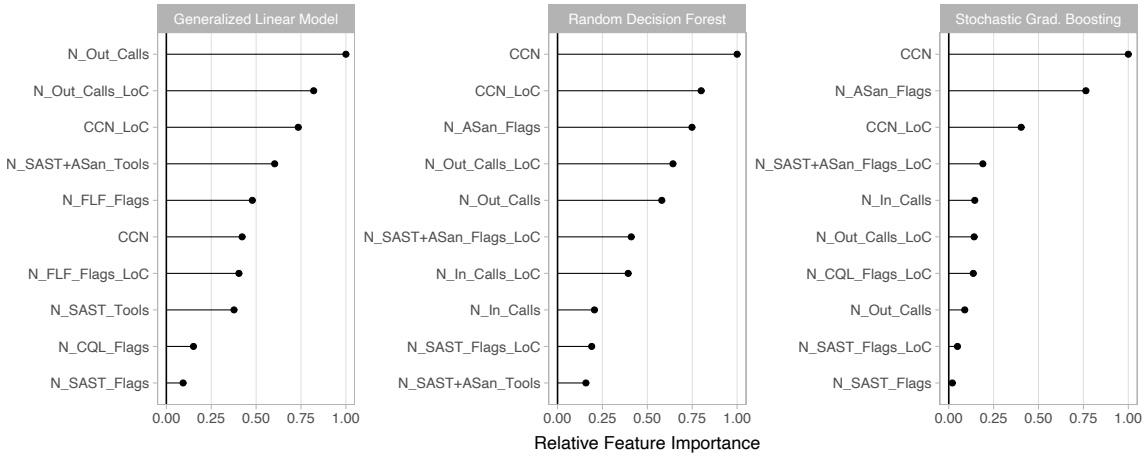


Figure 4.3.: Relative importance of the 10 most influential features in the different ML vulnerability prediction models.

functions in the training set. Each fold is thereby stratified, meaning that both prediction classes occur in each fold with the same probability as in the original non-downsampled dataset. Moreover, we repeat each cross-validation five times and average the results to provide a robust analysis of the models' effectiveness.

Feature Importance

Figure 4.3 shows the relative importance of the 10 most influential ML vulnerability prediction features used by GLM, RDF, and GBM. The figure thereby excludes FNN and SVM because their learned weights do not directly correspond to the feature importance. Also, note that the actual importance scores are normalized between 0 and 1 in order to represent the relative contribution of the feature to the predictions. A feature with a score of 0.5 is thus considered twice as important as one with a score of 0.25.

As shown in the figure, different models prioritize different features in their decision-making process. However, the complexity-based features, such as cyclomatic complexity number (feature `CCN`) and the number of outgoing function calls (`N_Out_Calls[_LoC]`), are often more important than the tool-based features in our setup. This suggests that code complexity is a stronger indicator of vulnerable functions than SAST tool- or AddressSanitizer-flagged code, at least at the function level. Yet, despite their lower importance, the tool-based features still provide helpful information that improves the models' prediction accuracy.

Furthermore, the integration of AddressSanitizer as an additional tool for identifying problematic code proves beneficial. In particular, the number of different tools (including AddressSanitizer) that flag a function (`N_SAST+ASan_Tools`) as problematic is more indicative of vulnerable functions than the same feature with only SAST tools (`N_SAST_Tools`), resulting in a 1.6 times higher impact in the GLM predictions. Moreover, the number of lines flagged only by AddressSanitizer (`N_ASan_Flags`) is among the top-three features in the RDF and GBM models, outperforming the next best tool-based feature (`N_SAST+ASan_Flags_LoC`) by about 1.8 and 3.8 times, respectively.

4.2.4. Application

Our vulnerability prediction approach can be easily applied in practice through the following steps, which are also visualized in Figure 2.2 on page 16.

First, the ML vulnerability prediction model must be trained on a dataset with known vulnerabilities, ideally using older, vulnerable versions of the same software that will later be targeted by the model (i.e., within-project prediction). The shared code characteristics allow the model to leverage project-relevant patterns, resulting in a better feature alignment and more accurate predictions of vulnerable functions [116, 338]. If no historical project data is available, the pre-trained models provided with this thesis (see Appendix A.1 on page 131) can be used at first to predict potentially vulnerable functions (i.e., cross-project prediction). However, for the reasons mentioned above, cross-project prediction is generally less effective than within-project prediction, as we will also see later in our evaluation.

Next, the feature values of each function in the SUT need to be extracted. For the tool-based features, this involves running the SAST tools and a modified version[6] of AddressSanitizer that outputs all instrumented code locations. Finally, the extracted values must be fed into the ML vulnerability prediction model (with the chosen cutoff) in order to determine the bug-prone functions.

4.3. Evaluation

In this section, we outline the setup for evaluating the effectiveness of our ML models in distinguishing between vulnerable and non-vulnerable functions. Moreover, we describe the evaluation results and discuss the main findings and insights.

4.3.1. Setup

Research Questions

This evaluation aims to verify whether our ML-based vulnerability prediction approach can reliably identify potentially vulnerable code for our proposed fuzzing metric. Therefore, we answer the following *research questions* through empirical evaluations:

RQ 2a: Within-Project Vulnerability Prediction. How effective are our ML vulnerability prediction models at identifying vulnerable functions in disjoint subsets of the functions in the same software projects used for model training?

RQ 2b: Cross-Project Vulnerability Prediction. How effective are our ML vulnerability prediction models at identifying vulnerable functions in completely different software projects than those used for model training?

Both research questions analyze the effectiveness of the trained models in two different scenarios in order to provide detailed insights about their applicability and limitations. Specifically, the first question examines the models' effectiveness on different versions or

[6] `https://github.com/tum-i4/llvm-project`

Table 4.2.: Subject programs employed for training and testing the ML vulnerability prediction models (as appeared in [170]).

(a) Training set

Subject	Version	# Lines	# Functions	# Vuln. Funcs.
Binutils *	2.29	45,243	1,378	15
FFmpeg	n3.3.2	486,774	17,759	23
Libpng	1.6.38	10,184	398	9
OpenSSL	3.0.0	165,274	13,036	30
PHP	8.0.0-dev	101,531	4,759	8
Poppler	0.88.0	61,081	4,458	20
SQLite3	3.32.0	53,327	2,296	16
Total		923,414	44,084	121

* We exclude NM, Objdump, and Size from Binutils to avoid training the models on the same functions they are tested on.

(b) Testing set

Subject	Version	# Lines	# Functions	# Vuln. Funcs.
LibTIFF	4.1.0	19,527	826	23
Libxml2	2.9.10	85,466	2,982	21
NM Objdump Size	2.29	90,566	2,792	50
Total		195,561	6,600	94

components of the same software projects on which they were trained. To avoid any bias from evaluating the models on functions that were also used for training, we repeatedly retain a rotating fold (of functions) from the training folds for testing the models. In contrast, the second question analyzes the models' effectiveness on a completely different set of software projects than those used for training. Ideally, both evaluations will yield positive results, demonstrating that our vulnerability prediction approach and, ultimately, our fuzzing metric can be applied independently of the SUT.

Training and Testing Programs

For training and testing the ML vulnerability prediction models, we randomly split the programs previously used for our SAST tool evaluation into two sets, as shown in Tables 4.2a and 4.2b. Together, these programs contain 215 vulnerable functions and 50,469 (likely) non-vulnerable functions. Note that we exclude NM, Objdump, and Size from the Binutils training programs because they are used for model testing. This way, we ensure that the models are not trained on the same functions that they will be tested on.

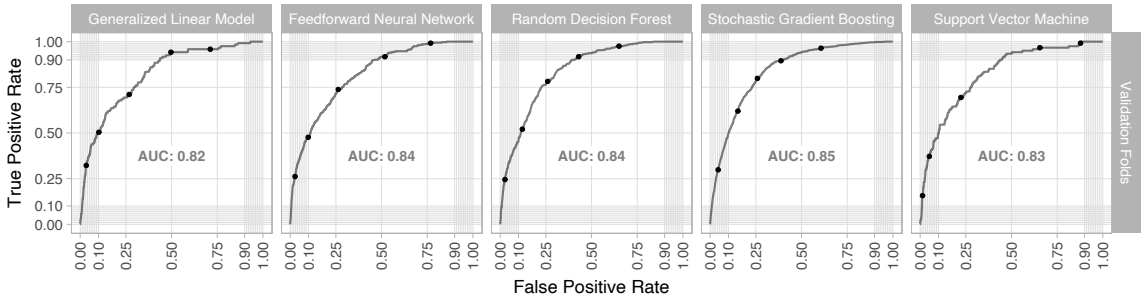


Figure 4.4.: Performance tradeoff of the ML vulnerability prediction models in identifying vulnerable functions across the validation folds of the training set (as appeared in [170]). The dots indicate the cutoff values $\theta \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, with the highest θ on the far left.

Metrics

To evaluate the effectiveness of the ML vulnerability prediction models, we use the true positive rate (TPR) and false positive rate (FPR) metrics defined in Equations (2.1) and (2.3) on page 14 and on page 17. We thereby consider a vulnerable function detected by the model if it correctly classifies the function as such.

Infrastructure

We performed all experiments on a machine with an Intel® Core™ i7-6700 CPU containing eight logical cores that run at 3.4 GHz, with access to 16 GB RAM and GNU/Linux Ubuntu 20.04 (64-bit) as the operating system. Using the `caret` R package [149], building the ML models in parallel on all cores took about 30 to 120 seconds, depending on the training algorithm.

4.3.2. Results

Besides the ML models described in Section 4.2.2 on page 53, we include a baseline model that randomly classifies functions based on the relative frequencies of vulnerable and non-vulnerable functions in our training set—i.e., $P(vuln) \approx 0.003$ and $P(non\text{-}vuln) \approx 0.997$ (see Table 4.2a on the preceding page). However, since the random classifier has a consistent ROC-AUC score of only 0.5 (no discrimination)[7], which indicates that the most vulnerable functions were not classified as such, we omit the corresponding ROC plots from the figures for brevity.

Within-Project Vulnerability Prediction

Figure 4.4 shows the performance tradeoff of the ML vulnerability prediction models in identifying vulnerable functions across the validation folds of the training set. The dots in the plots indicate the cutoff values $\theta \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, with the highest θ on the far left. The results indicate that the ML models effectively discriminate between

[7] We obtained similarly low ROC-AUC scores with a 50:50 random classification.

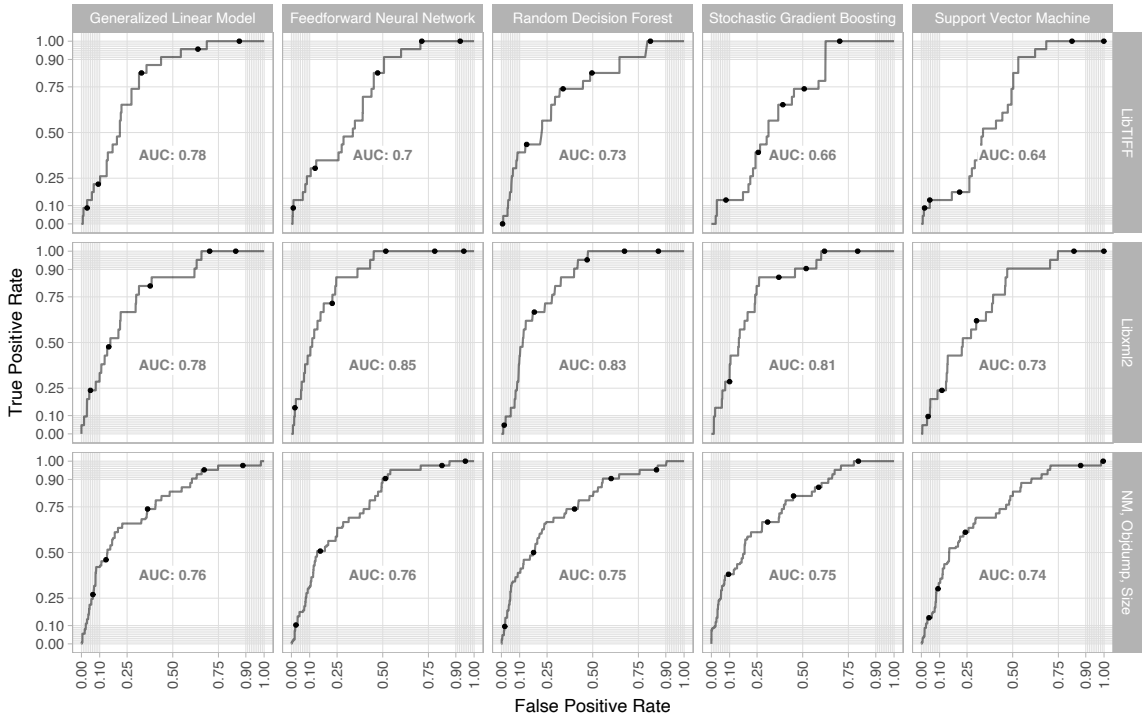


Figure 4.5.: Performance tradeoff of the ML vulnerability prediction models in identifying vulnerable functions in each testing program (as appeared in [170]). The dots indicate the cutoff values $\theta \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, with the highest θ on the far left.

vulnerable and non-vulnerable functions, achieving an average ROC-AUC score of about 0.84 (excellent discrimination). Thus, they not only outperform the random classifier but also reliably identify potentially vulnerable functions. Although the GBM achieves the highest score, the performance differences between the ML models are not statistically significant. Also, note that with a relatively low cutoff of 0.3, all five models achieve a TPR of 0.9 or higher, ensuring that they hardly miss any vulnerable functions. At the same time, their FPR (except for the SVM) remains below 0.5, indicating that, due to the few vulnerable functions in our dataset, about half of the functions classified as vulnerable are false positives.

Summary (RQ 2a)

Our evaluation shows that our ML vulnerability prediction models are very reliable in distinguishing between vulnerable and non-vulnerable functions in disjoint parts or older versions of the codebases used for model training (i.e., within-project prediction). Specifically, the trained models achieve an average ROC-AUC score of 0.84 across the validation folds of the training set, indicating excellent discrimination regardless of the employed training algorithm.

Cross-Project Vulnerability Prediction

Figure 4.5 on the preceding page is similar to the previous graph, but this time, it shows the performance tradeoff of the ML vulnerability prediction models in each testing program. As expected, the results are slightly lower than those obtained on the validation folds in the training set. Nonetheless, for NM, Objdump, and Size, which are treated as a single program due to their shared functions, the models achieve ROC-AUC scores ranging from 0.74 to 0.76, indicating a reasonable ability to identify (potentially) vulnerable functions. With respect to Libxml2, we observe a greater variability in model effectiveness—FNN, RDF, and GBM achieve excellent discrimination (ROC-AUC = 0.81–0.85). In contrast, GLM and SVM perform only moderately (ROC-AUC = 0.78 and 0.73, respectively). Similarly, for LibTIFF, GLM, FNN, and RDF effectively distinguish between vulnerable and non-vulnerable functions (ROC-AUC = 0.7–0.78). In contrast, GBM and SVM fall below the acceptable threshold [120], with ROC-AUC scores below 0.7.

Summary (RQ 2b)

Our evaluation shows that our ML vulnerability prediction models are slightly less effective in distinguishing between vulnerable and non-vulnerable functions in programs outside the training set (i.e., cross-project prediction). Nonetheless, the trained models achieve an average ROC-AUC score of 0.75 on the testing set, indicating reasonably good discrimination, albeit at the cost of more false positives.

4.3.3. Discussion

Our feature importance analysis shows that the impact of the selected features on the vulnerability prediction varies significantly among the ML models. Interestingly, in our setup, features derived from code complexity metrics are often more predictive of vulnerable functions than those derived from SAST tool findings. Nonetheless, features from *both* categories are necessary to achieve the best possible accuracy in vulnerability prediction. The final feature set allows for reliable and robust predictions with minimal performance differences between the various ML vulnerability prediction models, thus simplifying the selection of an appropriate model.

Although our vulnerability prediction models reliably identify potentially vulnerable functions in programs outside the training set, we recommend either (i) training such models on older, vulnerable versions of the same software project(s) that will later be targeted by fuzzing or (ii) starting with the pre-trained ML models[8] provided with this thesis (see Appendix A.1 on page 131) and then improving them over time as more data about vulnerabilities in the targeted projects becomes available. This will allow the models to take advantage of project-specific code characteristics, ultimately leading to more accurate predictions.

This is also supported by our evaluation, where the models' ROC-AUC scores are higher in the validation folds of the training set than for the functions in the testing set. For

[8] For example, the RDF or GBM model with a conservative cutoff value of 0.3 to 0.5 can be used as a starting point to minimize the risk of missing vulnerable functions.

example, in our within-project evaluation, a cutoff of 0.3 leads to a TPR of 0.9 or higher in all five ML vulnerability prediction models, and FPR of mostly below 0.5. Thus, given the small number of vulnerable functions in our subject programs, fuzzing only half of the functions could potentially trigger almost all of the contained vulnerabilities. Accordingly, the more effective the employed model, the more accurately our vulnerability-oriented code coverage metric will reflect fuzzing performance.

4.4. Threats to Validity

To minimize threats to *external validity*, we used 27 real-world programs with a total of 215 vulnerable functions, as well as five popular ML training algorithms to train and test our vulnerability prediction models.

In terms of *internal validity*, we randomly divided the subject programs into disjoint training and testing sets to avoid potential bias from program cherry-picking or overlapping training and testing functions. Also, to counteract overfitting to the majority output class of our unbalanced dataset, we followed the best practices for ML-based vulnerability prediction, including downsampling and repeated stratified cross-validation.

4.5. Summary

In this chapter, we explored the integration of code complexity metrics and SAST tool findings into ML vulnerability prediction models to identify potentially vulnerable code more reliably than with SAST tools alone. Our evaluation shows that the trained models are very effective at distinguishing between vulnerable and non-vulnerable functions in both within-project and cross-project scenarios. In the next chapter, we use these vulnerability prediction models for our vulnerability-oriented code coverage metric in order to stop fuzzing campaigns more (cost-)efficiently compared to using regular code coverage.

Part III.

Improving Accuracy in Reflecting Fuzzing Performance

5. Real-Time Coverage Tracking

This chapter presents the implementation, application, and practical benefits of our fuzzer-independent coverage analyzer, which allows real-time monitoring and, thus, dynamic termination of fuzzing campaigns. It also describes a comprehensive coverage dataset created with our tool that can be used for further empirical fuzzing research. Parts of this chapter have appeared in a peer-reviewed publication [169], first-authored by the author of this thesis.

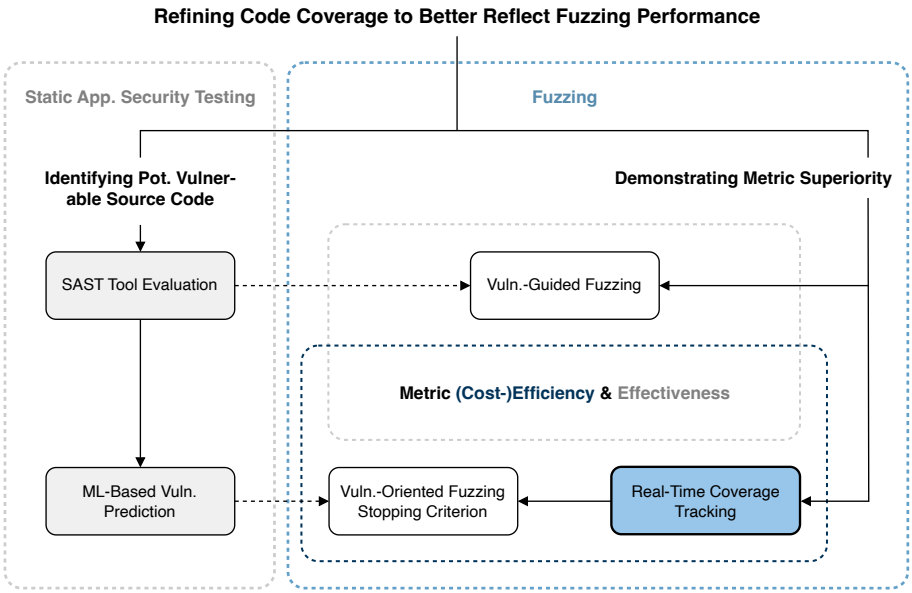


Figure 5.1.: Big Picture (Contribution 3: Real-Time Coverage Tracking).

5.1. Introduction

Context. As shown in Figure 5.1, this chapter belongs to the part of this dissertation where we demonstrate the improved accuracy of our vulnerability-oriented code coverage metric over regular code coverage in reflecting fuzzing performance. To this end, the next chapter evaluates the achievable time savings (i.e., the (cost-)efficiency) when using our metric to stop fuzzing campaigns based on their progress in covering potentially vulnerable code. Before that, however, we present the technical solution for tracking the necessary code coverage information in real-time during fuzzing so that our vulnerability-oriented stopping criterion can be applied in practice.

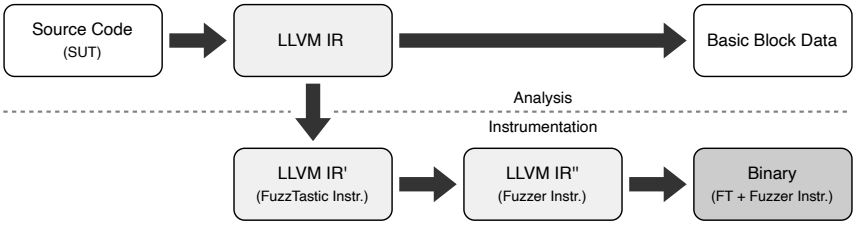


Figure 5.2.: Analysis (top) and instrumentation artifacts (bottom) required (left) and generated by FuzzTastic (right) (as appeared in [169]).

State-of-Practice and Problem. Existing technical solutions [2, 164, 189, 190] for collecting code coverage information involve *sampling* a representative subset of all fuzz inputs generated during the campaign and then *replaying* them on a second instance of the SUT that is compiled with coverage tracking support [1, 239]. However, this approach is limited to fuzzers that persist the performance-enhancing inputs in a dedicated directory representing the fuzzer queue so that a coverage analyzer can access them. In addition, replaying the stored inputs in real-time to monitor campaigns adds significant hardware and synchronization overhead, making it impractical for large-scale fuzzing.

Solution Approach. To address this (technical) research gap, we developed a new *real-time* code coverage tracking tool that is compatible with any blackbox, greybox, or whitebox fuzzer. Specifically, we first instrument the SUT at compile time to track the lines, BBs, and functions covered during fuzzing, storing the code coverage data in a shared array that is accessible by our tool. Throughout the campaign, our coverage analyzer then polls this array at intervals, thereby persisting the current coverage data into separate report files for further processing. We demonstrate the applicability of our analyzer by generating a comprehensive dataset of detailed coverage data from 9 different fuzzers executed on 12 real-world programs, which we use in the next chapter to evaluate the (cost-)efficiency of our fuzzing metric.

Contributions. Overall, we contribute a fuzzer-independent coverage analyzer that supports real-time code coverage tracking of all fuzz inputs generated throughout the campaign, which is required to dynamically end fuzzing based on a stopping criterion. As part of this contribution, we generate with our analyzer a large multipurpose dataset that contains extensive coverage data worth over 12 CPU-years of fuzzing, which can also be used for fuzzing research outside of this thesis.

5.2. The FuzzTastic Analyzer

In this section, we introduce our coverage analyzer, called FuzzTastic. Specifically, we outline the required and generated software artifacts, as well as its approach for tracking code coverage information in parallel with the fuzzing campaign.

```
{
    "basic_blocks": [
        {
            "id": 0,
            "file": "gif2png.c",
            "function": "interlace_line",
            "lines": [45, 48, 50]
        }, ...
    ]
}
```

(a) BB Metadata

```
{
    "timestamp": {
        "start" : "2021-04-22 14:25:19",
        "report": "2021-04-22 16:25:21"
    },
    "block_coverage": [
        79704586,
        ...
    ]
}
```

(b) BB Coverage

Listing 5.1.: Example of the basic block (BB) metadata file (left) and the corresponding coverage file (right) generated by FuzzTastic (both in JSON format) when fuzzing Gif2png with AFL for two hours (as appeared in [169]). During this time, the BB with the ID 0 was hit 79,704,586 times.

5.2.1. Program Analysis and Instrumentation

Figure 5.2 on the facing page shows the software artifacts involved when using FuzzTastic. Starting with the source code of the SUT, we first need to translate it into LLVM's intermediate representation (IR)[1], which we then analyze and instrument using a custom LLVM compiler pass [241]. FuzzTastic's compiler pass then statically analyzes the IR file, extracts the code locations of the BBs in the SUT, and stores this metadata in a JSON file. Specifically, these properties include for each BB (i) a unique numeric identifier (`id`), (ii) the path to the source file containing the BB (`file`), (iii) the name of the function containing the BB (`function`), and (iv) the line numbers enclosing the BB (`lines`).

An example of such metadata for the program Gif2png is shown in Listing 5.1a. In this listing, the BB with the identifier 0 (see line 4) spans the source code lines 45 to 50 (where the lines 46, 47, and 49 are empty) in the function `interlace_line`, which is defined in the source file `gif2png.c`.

```
static int32_t ft_shm_data[<N_BASIC_BLOCKS>];

void __ft_inc_cov(int32_t bb_id) {
    ft_shm_data[bb_id] += 1;
}
```

Listing 5.2.: Instrumentation employed by FuzzTastic to track basic block (BB) coverage (as appeared in [169]).

[1] This refers to a low-level, language- and platform-independent instruction set used by the LLVM compiler framework to perform optimizations and to generate machine code [153].

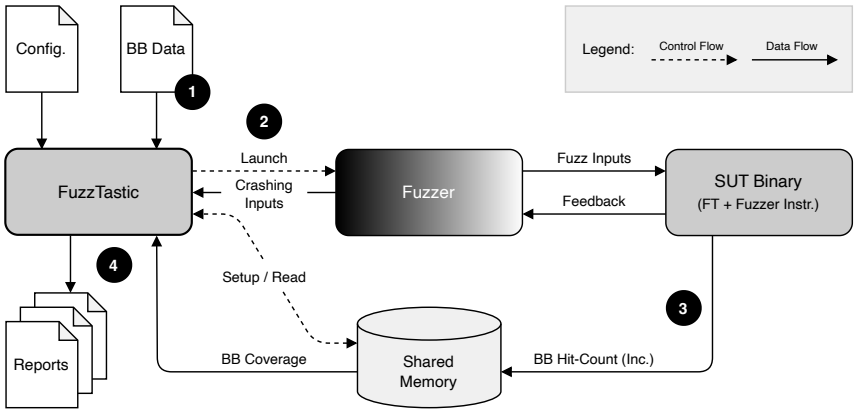


Figure 5.3.: Big picture showing the control and data flow between FuzzTastic, the selected blackbox, whitebox, or greybox fuzzer, and the instrumented system under test (SUT) to track basic block (BB) coverage information during a fuzzing campaign (adapted from [169]).

5.2.2. Code Coverage Tracking

Parallel to the static analysis, each BB is instrumented with a function call to `__ft_inc_cov` in Listing 5.2 on the preceding page, which is stored in a dynamic library. Thus, each time a BB is executed, this function increments the corresponding hit counter, holding the total number of executions in the array `ft_shm_data` (see line 3 in the listing), which, in turn, is stored in a *shared* memory segment[2]. This allows FuzzTastic to efficiently and accurately[3] track the code coverage of *all* generated fuzz inputs, not just those in the fuzzer queue [164, 189, 190]. Finally, the instrumentation of the employed fuzzer must be applied, followed by compiling the instrumented IR file into a fuzzable binary.

During fuzzing, the current code coverage is then repeatedly queried and stored in separate report files. An example of such a coverage report, generated after two hours of fuzzing the Gif2png with the AFL fuzzer, is shown in Listing 5.1b on the previous page. In this period, the BB 0 (see line 7 in the listing) was executed 79,704,586 times, while the least frequently executed BB was hit only 2,353 times (not shown in the listing). Note that in addition to BB coverage, the generated reports also allow measuring line and function coverage since the BB metadata file provides information about both code locations.

5.2.3. Application

Figure 5.3 shows the interactions between FuzzTastic, the selected fuzzer, and the instrumented SUT in order to collect BB coverage information during a fuzzing campaign. This involves the following steps:

[2]This form of inter-process communication (IPC) allows multiple processes to directly access and modify the same designated memory region, which enables rapid data exchange [280, pp. 303–323].

[3]Unlike the replay approach, ours accurately measures code coverage even for fuzzers that hash coverage information using *collision-prone* hashing functions [86, 89, 299], which can produce identical hash values for different execution paths [63]. Colliding coverage-increasing inputs are then not added to the queue and are thus missed by the replay-based coverage analysis [169].

In **step** ❶, FuzzTastic takes as input the extracted BB metadata and a user-defined fuzzing configuration, which is written as bash scripts. This configuration specifies the paths to the (instrumented) SUT binary, the fuzzer, and the seed corpus directory, along with their respective command-line interface (CLI) options and environment variables. Moreover, it specifies the campaign duration, the fuzzer's working directory, and the directory where code coverage reports should be stored.

In **step** ❷, FuzzTastic sets up a shared memory segment and allocates an array sized according to the number of BBs contained in the SUT, in which the coverage information will later be stored. After that, FuzzTastic launches the fuzzer, which runs the campaign with the given configuration.

In **step** ❸, while the fuzzer continues to generate fuzz inputs, the instrumentation increments the hit count of each BB in the shared coverage array that was covered during execution. The array indices are thereby aligned with the BB identifiers in the metadata file so that the coverage information can be mapped to the corresponding BB code locations. Hence, this array contains the code coverage achieved by the fuzzer at any time during the campaign.

In **step** ❹, FuzzTastic repeatedly retrieves the current code coverage from the array at a user-defined interval (15 minutes by default) until the campaign is stopped, thereby storing this data in separate report files. Thus, a new report is generated after each interval, which allows for in-depth coverage analyses in real-time.

5.3. Generating a Coverage Dataset

While FuzzTastic was primarily developed to monitor fuzzing campaigns in order to determine optimal stopping points, we now use it to generate a comprehensive coverage dataset (see Appendix A.1 on page 131). Thereby, we demonstrate its ability to track consistent code coverage information across different fuzzers and real-world programs. In the next chapter, we will then use this dataset to evaluate the accuracy with which our metric measures fuzzing (cost-)efficiency.

5.3.1. Subject Programs

For our coverage dataset, we looked for subject programs that (i) are widely used in practice to ensure that the dataset accurately reflects real-world fuzzing and (ii) span different domains to allow for more generalizable evaluations. The final selection comprises the 12 programs shown in Table 5.1 on the following page, including tools for binary manipulation (NM, Objdump, Readelf, Size, and Strings from the Binutils suite), data processors and converters for various file formats (FFmpeg, FreeType2, Gif2png, JasPer, and JsonCpp), network utilities (Libpcap), and data compression tools (Zlib). To fuzz the libraries, we use the fuzz harnesses provided by their repositories or by the OSS-Fuzz platform [264].

Table 5.1.: Subject programs employed for creating a fuzzing coverage dataset using the FuzzTastic tool (adapted from [169]).

Subject	Version	# Lines	# Basic Blocks	# Functions
FFmpeg	n3.3.2	522,813	432,244	21,147
FreeType2	2.7	44,686	27,521	1,635
Gif2png	2.5.3	988	700	27
JasPer	1.900.0	17,385	14,417	720
JsonCpp	1.8.4	7,251	5,938	1,328
Libpcap	1.9.0	12,076	6,442	497
NM	2.29	68,667	44,114	2,126
Objdump	2.29	89,961	60,448	2,701
Readelf	2.29	22,347	18,578	477
Size	2.29	68,115	43,711	2,101
Strings	2.29	68,048	43,714	2,093
Zlib	1.2.9	4,223	3,289	148
Total		926,560	701,116	35,000

5.3.2. Fuzzers

To demonstrate FuzzTastic's compatibility with various fuzzers, we selected the following tools that have been published at top research conferences or are widely used according to their GitHub stars [28]:

AFL (2.56b): This coverage-based greybox fuzzer uses compile-time instrumentation to collect test execution and coverage information at runtime, which, in turn, guides the generation of fuzz inputs. Inputs that trigger a crash or increase code coverage are added to the queue for further mutation. Based on the execution time and branch coverage of the queued inputs, AFL assigns an *energy score* that determines their mutation time for the next mutation cycle [326].

AFLFast (2.51b): This fuzzer is an advanced fork of AFL that models path transition probabilities as a *Markov chain* [217] to increase code coverage more effectively. Using this probabilistic model, AFLFast incorporates *adaptive power schedules* (and smarter seed selection) that prioritize paths that are rarely or not yet executed by assigning more energy to seeds that execute low-frequency paths, and vice versa [25].

AFLSmart (2.52b): This fuzzer is an advanced fork of AFL that uses a *validity-based* power schedule and *structure-preserving* input mutations to reach deeper code regions in the SUT. The schedule assigns more energy to seeds whose mutations are more likely to pass the input parser. For test generation, AFLSmart leverages the high-level structure of the seed rather than its raw bytes to perform structure-aware mutations, ensuring the validity of new fuzz inputs [235].

AFL++ (2.64c): This fuzzer is an advanced, *community-driven* fork of AFL that incorporates the latest fuzzing research. For instance, AFL++ includes (i) collision-free coverage instrumentation, (ii) a custom interface for new mutation operators, (iii) mechanisms for overcoming complex comparisons, (iv) AFLFast's adaptive power schedules, and (v) snapshot-based process restoration for more efficient fuzzing [86].

Eclipser (1.0): This fuzzer is essentially a concolic testing engine that cleverly alternates between lightweight whitebox and classical greybox fuzzing in order to maximize code coverage. In doing so, Eclipser approximates path constraints solvable by *generational search* [97], rather than relying on the computationally expensive satisfiability modulo theories (SMT) solving [201] employed in classical dynamic symbolic execution [53].

FairFuzz (2.52b): This fuzzer is an advanced fork of AFL that incorporates novel mutation operators specifically designed to generate inputs that exercise branches that guard code that is empirically hard to cover. To achieve this, FairFuzz implements a dynamic *mutation masking* technique that identifies crucial parts of an input that must remain unchanged to ensure that the input hits the rare branches [157].

Honggfuzz (2.2): This coverage-based greybox fuzzer generally follows a similar approach to AFL. However, unlike most other fuzzers, Honggfuzz supports low-level *process tracing*, enabling robust crash detection by intercepting hijacked signals that might otherwise be hidden during execution. It also supports hardware-based execution feedback, allowing the collection of detailed code coverage information with minimal performance overhead [99].

MOpt-AFL[++] (2.52b/2.64c): These fuzzers are advanced forks of AFL and AFL++ that use a fuzzer-independent *mutation scheduling* framework called MOpt. Unlike the static scheduling in most mutation-based fuzzers, MOpt dynamically adjusts the schedule using a custom particle swarm optimization (PSO) algorithm [77]. This algorithm aims to find an optimal strategy for selecting mutation operators based on their efficiency in discovering interesting fuzz inputs [175].

5.3.3. Seeds, Duration and Repetitions

For each subject-fuzzer pair in our setup, we run fuzzing campaigns with an *empty* seed and a *non-empty* seed corpus to include code coverage data that show their impact on the fuzzing performance. For the empty seed, we use the smallest amount of data that the subject programs' input parser accepts (e.g., `{}` for JsonCpp), whereas for the non-empty corpus, we use the seeds provided by AFL's repository.

To capture performance peaks of fuzzers that become more effective as more seeds are added to the queue, we set the campaign length to 24 hours. We also repeat each campaign 20 times to allow studies using this dataset to account for the randomness in fuzzing through statistical tests.

5.3.4. Infrastructure

Since building a dataset with the setup described above requires about 12 CPU-years of fuzzing, we performed all experiments on the Leibniz Supercomputing Centre (LRZ) high-performance computing (HPC) cluster[4], equipped with Intel® Xeon® E5-2690v3 CPU nodes. Each node contains 28 physical cores that run at 2.6 GHz, with access to 64 GB RAM and SUSE Linux Enterprise Server (SLES) 15 (64-bit) as the operating system. On this cluster, we distributed the campaigns across 96 nodes, corresponding to 2688 cores and 6144 GB RAM. Each campaign was assigned a dedicated core and 4 GB RAM, with the fuzzer working directory located on a RAM drive to avoid I/O bottlenecks.

5.4. Technical Limitations

The path-collision-free coverage tracking implemented in FuzzTastic comes with the limitation that extremely large programs may not be analyzable due to the large amount of RAM required by the array storing the coverage information. However, as demonstrated when creating the dataset, even large programs like FFmpeg with 432,244 BBs could be analyzed using a (virtual) machine with only 4 GB RAM, suggesting that FuzzTastic is practical for most programs.

Furthermore, FuzzTastic's code instrumentation—like that of other coverage analyzers [1, 239]—induces some runtime overhead that reduces the program execution speed. However, by instrumenting at the less fine-grained level of BBs and using efficient shared memory IPC, we minimized this overhead as much as possible.

The final limitation concerns the fuzzing of multi-threaded programs. Here, FuzzTastic's implementation to track code coverage is susceptible to data races, which can lead to deviations in BB hit counts. However, we consider this a reasonable tradeoff, as adding any synchronization between threads would increase the runtime overhead and potentially negatively affect the campaign's performance.

5.5. Summary

In this chapter, we introduced FuzzTastic, a fine-grained coverage analyzer that can be attached to any fuzzer in order to track the exact lines, BBs, and functions covered by all fuzz inputs generated during a fuzzing campaign. Unlike existing approaches, it captures code coverage in real-time (i.e., in parallel with the campaign), e.g., allowing campaigns to be terminated dynamically based on a stopping criterion rather than using a fixed time budget. We also described the large coverage dataset created with FuzzTastic, which we use in the next chapter to demonstrate the improved (cost-)efficiency of our fuzzing metric over regular code coverage.

[4] `https://doku.lrz.de/display/PUBLIC/High+Performance+Computing`

6. Vulnerability-Oriented Fuzzing Stopping Criterion

This chapter presents a novel stopping criterion that terminates fuzzing campaigns based on the saturation of our vulnerability-oriented code coverage metric. By analyzing the tradeoff between fuzzing time saved and security bugs missed, compared to the saturation of program crashes and regular code coverage, we demonstrate the superior accuracy of our metric in reflecting fuzzing (cost-)efficiency. Parts of this chapter have appeared in a peer-reviewed publication [170], first-authored by the author of this thesis.

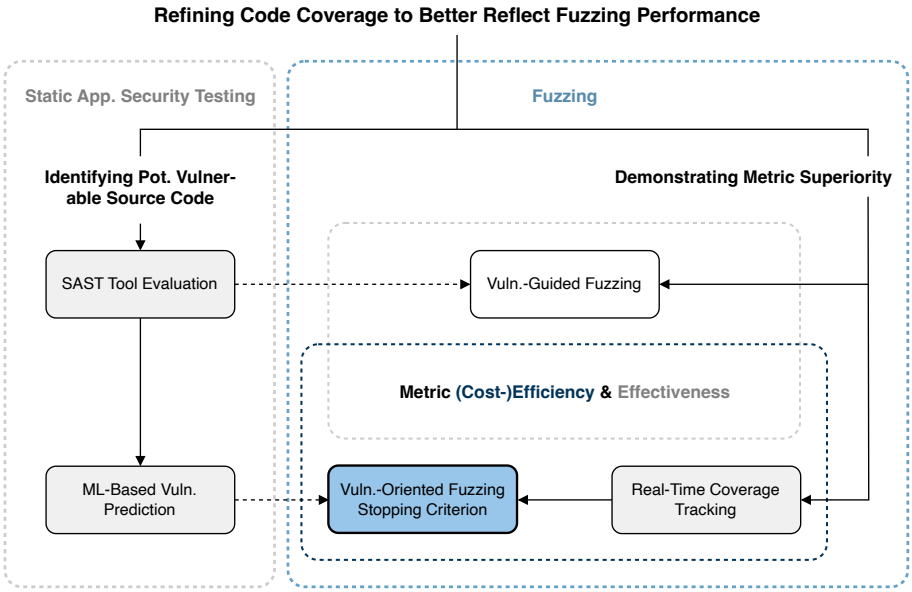


Figure 6.1.: Big Picture (Contribution 4: Vuln.-Oriented Fuzzing Stopping Criterion).

6.1. Introduction

Context. As shown in Figure 6.1, this chapter belongs to the part of this dissertation where we demonstrate the improved accuracy of our vulnerability-oriented code coverage metric in reflecting fuzzing (cost-)efficiency over the number of triggered program crashes and regular code coverage. For this, we use these metrics to stop fuzzing campaigns when the respective metric score saturates for a longer period, thereby comparing their cost-efficiency, i.e., the tradeoff in terms of saved fuzzing time versus missed security bugs. Hence, this chapter also proposes a novel stopping criterion for campaigns.

State-of-Practice and Problem. Better stopping criteria is an important but under-explored area of research, given the *high* resource demands of fuzzing [22]. Currently, campaigns are stopped when (i) a fixed fuzzing time budget has expired [144, 177, 258], (ii) no progress in crash generation or code coverage is observed for an extended period [284, pp. 437–470, 285, pp. 71–77, 287], or (iii) the bug-discovery probability falls below a certain threshold [23]. However, the metrics used in (i) and (ii) often overestimate fuzzing effectiveness, keeping campaigns alive even though they only trigger redundant crashes or cover non-critical code. Existing approaches for (iii) are fuzzer-dependent because they must account for the *adaptive bias* [20, 23] in fuzzing strategies where the probability of a bug-revealing fuzz input increases as the campaign progresses.

Solution Approach. To address this research gap, we use our metric as a *saturation-based stopping criterion* for fuzzing campaigns. We hypothesize that significant time savings can be achieved with minimal risk of missing bugs by dynamically terminating campaigns when they stagnate on the potentially vulnerable code. The strategy is as follows: (i) before fuzzing, the SUT is statically analyzed for bug-prone code using a ML vulnerability prediction model (see Chapter 4 on page 49); (ii) during fuzzing, the intersection between the identified problematic code and the code covered by the fuzz inputs is calculated; (iii) the campaign is stopped if the set of covered, potentially vulnerable code (i.e., our metric's score) has not increased for a certain period. Otherwise, the strategy returns to step (ii). We thereby use the FuzzTastic dataset (see Section 5.3 on page 69) to evaluate the cost-effectiveness of our approach.

Contributions. Overall, we contribute a vulnerability-oriented stopping criterion for fuzzing campaigns based on the saturation of our metric. Our evaluation shows that, compared to the saturation of program crashes and regular code coverage, using our criterion to stop 24-hour campaigns can save, on average, 6–12 hours of fuzzing time while missing fewer than 0.5 out of 12.5 security bugs. Thus, these results demonstrate the improved accuracy of our metric in reflecting fuzzing (cost-)efficiency.

6.2. Motivation

Using the number of program crashes is very susceptible to under- or overestimating fuzzing performance [144]. When used as a saturation-based stopping criterion, this can lead to missed security bugs or unnecessarily long fuzzing campaigns. Since code coverage reduces the risk of underestimation, it is often preferred in practice. However, it still tends to overestimate the effectiveness of campaigns because regular coverage assumes that all code regions are equally important and, therefore, equally vulnerable.

In the following, we discuss the shortcomings of these two stopping criteria using concrete campaign examples and theoretically demonstrate how our criterion mitigates these problems.

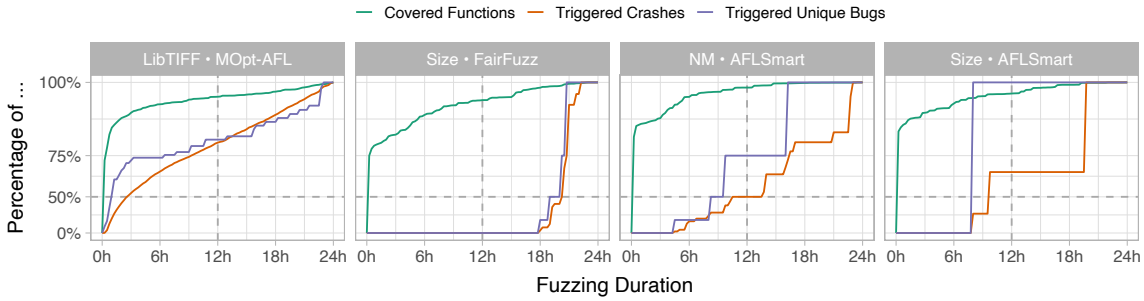


Figure 6.2.: Examples of accurate estimation of fuzzing effectiveness by program crash count and regular function coverage (first graph), under- and overestimation by crash count (second and third graphs), and overestimation by function coverage (fourth graph) (as appeared in [170]). The plotted scores represent the arithmetic means across 20 fuzzing campaign repetitions.

Saturation of Program Crashes. Crash saturation can be a good stopping criterion if the course of triggered crashes closely resembles that of the unique bugs, as shown in the first graph of Figure 6.2, for example. However, it loses accuracy when crashes occur very late in the campaign or are repeatedly caused by the same underlying bug(s). The first scenario is shown in the second graph, where FairFuzz does not trigger any crashes for almost 18 hours. Here, the campaigns would be terminated prematurely, resulting in several bugs being missed. In contrast, the second scenario is shown in the third graph, where AFLSmart discovers all detectable bugs in NM within 16 hours, yet the number of crashes continues to increase for another seven hours. Such a large number of crashes can give a false sense of fuzzing progress, resulting in unnecessarily long campaigns.

Saturation of Regular Code Coverage. For this reason, code coverage is often used as a (complementary) measure to better decide when to stop fuzzing. Returning to the Size-FairFuzz example, although no crashes are triggered during the first 18 hours of fuzzing, the function coverage steadily increases until the campaigns reach the vulnerable code of the SUT. Also, in the example where NM is fuzzed with AFLSmart, the campaigns begin to saturate on the same functions around the time all detectable bugs are found, providing more accurate feedback about the fuzzing performance than the number of triggered crashes. Accordingly, code coverage can help avoid missing bugs and save fuzzing time when used as a saturation-based stopping criterion.

Nonetheless, code coverage is still very much prone to overestimation. For example, a campaign that keeps increasing code coverage is considered very effective and is, therefore, not terminated, regardless of whether or not any new security-critical code has been reached. This case is shown in the fourth graph, where AFLSmart finds all detectable bugs in Size after about 7.5 hours of fuzzing, yet the function coverage increases on non-critical functions for another 11.5 hours. To end such long campaigns earlier, we suggest checking the saturation of our vulnerability-oriented code coverage metric, i.e., only the covered, potentially vulnerable code identified by a ML vulnerability prediction model instead of the entire code.

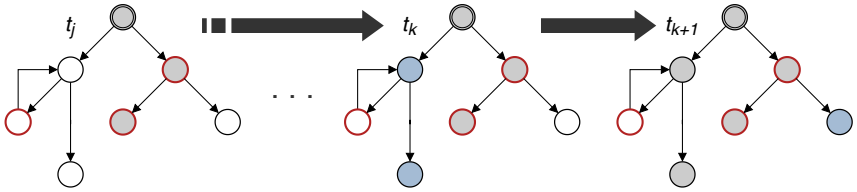


Figure 6.3.: Visualization of our hypothesis at the time points t_j, t_k, and t_{k+1} during fuzzing (as appeared in [170]). The gray and blue nodes are newly covered code regions, and the red-outlined nodes are bug-prone regions.

Saturation of Covered, Potentially Vulnerable Code. A simplified example of our hypothesis that significant fuzzing time can be saved by stopping campaigns when they saturate on the potentially vulnerable code is visualized in Figure 6.3. It shows the CG or CFG of a SUT with the covered code regions (grey nodes) at three different times, t_j, t_k (where $j < k$), and t_{k+1}, during fuzzing. In this example, the coverage increases by two code regions from t_j to t_k and by one region from t_k to t_{k+1} (blue nodes), indicating that the campaign remains active and should therefore not be stopped. However, the coverage of potentially vulnerable code regions (red-outlined nodes) does not increase at any point. At t_k, the campaign appears to soon cover the last bug-prone region (which may not even be reachable from the fuzz entry), but at t_{k+1}, the execution shifts in a different direction, covering rather irrelevant code. Thus, if we use vulnerability-oriented code coverage instead of regular code coverage as a saturation-based stopping criterion, we can probably already terminate the campaign at t_k or earlier, depending on how long a campaign is allowed to stagnate before termination.

6.3. Vulnerability-Oriented Stopping Approach

In this section, we first provide a general definition of saturation-based stopping criteria for fuzzing campaigns. Then, we introduce a new criterion based on our fuzzing metric that aims to terminate campaigns more cost-efficiently than existing criteria.

6.3.1. Saturation Analysis

Fuzzing campaigns are typically stopped based on the *progress* of the campaign's effectiveness as measured by some metric. This dynamic approach, as opposed to using a fixed time budget, provides better utilization of the fuzzing resources—it avoids wasting valuable resources by terminating ineffective campaigns early and reduces the risk of missing bugs by extending the duration of effective campaigns.

Definition 6.1 (Saturation Window)**.** A saturation window δ specifies the time period during which the effectiveness of a fuzzing campaign, as measured by a metric S, must stagnate before the campaign is terminated.

Definition 6.2 (Allowable Metric Deviation)**.** An allowable deviation ϵ specifies the relative upper limit to which the score of a fuzzing metric S may increase during the saturation window δ so that the fuzzing campaign is still considered stagnant, i.e., $S(t + \delta) \leq S(t) + \epsilon$.

Based on the chosen fuzzing metric, the saturation window (see Definition 6.1 on the facing page), and the allowable metric deviation (see Definition 6.2 on the preceding page), we define saturation-based stopping criteria for fuzzing campaigns as follows.

Definition 6.3 (Saturation-Based Stopping Criterion)**.** A saturation-based stopping criterion for fuzzing campaigns is defined as a triple (S, δ, ϵ), where:

- S is the metric for quantifying the fuzzing effectiveness,
- δ is the saturation window, and
- ϵ is the allowable deviation of S within δ.

After t_k time units, a campaign is then stopped if and only if $t_k - t_j = \delta$, where $j < k$ and t_j denotes the time since the campaign started to stagnate, and $S(t_k) - S(t_j) \leq \epsilon$.

Definition 6.3 provides a generic definition of saturation-based stopping criteria. To apply it in practice, the parameters S, δ, and ϵ must be instantiated with a specific fuzzing metric and concrete values for the saturation window and allowable metric deviation. Thereby, the more accurately the employed metric reflects fuzzing effectiveness, the better the campaign's optimal termination point can be determined.

6.3.2. Vulnerability-Oriented Code Coverage

Since regular code coverage treats all code regions as equally important, it is prone to misleading fuzzing assessments. To address this, we propose *vulnerability-oriented code coverage*—a coverage variant that focuses on the potentially vulnerable code covered during fuzzing. To define our metric, we first introduce the function $\texttt{pot\_vuln} : \mathcal{D} \times \mathcal{R} \to \{\mathit{true}, \mathit{false}\}$, which returns, from all code regions $R \in \mathcal{R}$ in the SUT, those that are flagged as problematic by the employed tools $\mathfrak{D} \in \mathcal{D}$:

$$\texttt{pot\_vuln}(R, \mathfrak{D}) \coloneqq \bigcup_{\mathfrak{d} \in \mathfrak{D}} \texttt{analyze}(\mathfrak{d}, R) \tag{6.1}$$

Note that the function in Equation (6.1) runs each tool $\mathfrak{d} \in \mathfrak{D}$ using the analysis function in either Definition 2.1 or Definition 2.4, depending on whether it is a SAST tool or a ML vulnerability prediction model.

Definition 6.4 (Vulnerability-Oriented Code Coverage)**.** Using the function in Equation (6.1) with the vulnerability prediction tools $\mathfrak{D} \in \mathcal{D}$, we define our vulnerability-oriented code coverage metric $S_{\text{vuln-cov}} : \mathcal{R} \times \mathcal{D} \times T \to \mathbb{R}_{[0,1]}$ for the code regions $R \in \mathcal{R}$ in the SUT after $t \in T$ time units within the fuzzing campaign as follows:

$$S_{\text{vuln-cov}}(R, \mathfrak{D}, t) \coloneqq \frac{\mathbf{card}\Big(\big\{r \mid r \in \texttt{pot\_vuln}(R, \mathfrak{D}) \wedge \texttt{covered}(r, t)\big\}\Big)}{\mathbf{card}\big(\texttt{pot\_vuln}(R, \mathfrak{D})\big)}$$

where $\mathbf{card}(\cdot)$ returns the set cardinality.

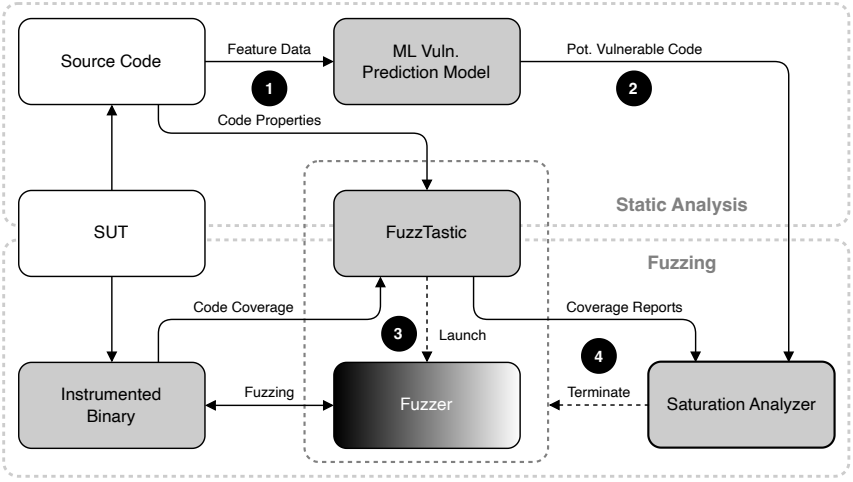


Figure 6.4.: Big picture showing the control (dashed arrows) and data flow (solid arrows) between the system under test (SUT), the machine learning (ML) vulnerability prediction model, FuzzTastic, and the saturation analyzer to terminate a fuzzing campaign according to our stopping criterion.

Our vulnerability-oriented code coverage metric defined in Definition 6.4 on the previous page provides more detailed information about a campaign's potential to find security bugs. For example, a campaign that covers 50% of the entire source code may exercise 80% or only 20% of the security-critical code. However, the reliability of our metric *depends* on how effectively bug-prone code can be statically identified.

To derive our *vulnerability-oriented stopping criterion*, we instantiate the generic saturation-based stopping criterion in Definition 6.3 on the preceding page with our fuzzing metric. Thereby, we suggest using the ML-based vulnerability prediction approach described in Chapter 4 on page 49 to identify the potentially vulnerable code before fuzzing.

As for the saturation window and the allowable metric deviation, we explore the impact of different values on the cost-effectiveness of our stopping criterion in Section 6.4 on the facing page. This allows us to make recommendations for these two parameters for the practical application of our criterion.

6.3.3. Application

Figure 6.4 shows the interactions between the SUT and the various tools in order to dynamically terminate a fuzzing campaign according to our vulnerability-oriented stopping criterion, which includes the following steps:

In **step** ❶, the feature data and code properties required by the employed ML vulnerability prediction model (see Section 4.2.1 on page 51) and the FuzzTastic coverage analyzer (see Section 5.2.1 on page 67), respectively, must be extracted from the source code of the SUT.

In **step** ❷, the extracted feature data need to be fed into the prediction model to identify the potentially vulnerable code regions (see Section 4.2.4 on page 57).

In **step** ❸, the workflow transitions from static analysis to fuzzing. Here, FuzzTastic is started with the extracted code properties and a specific fuzzing configuration (not shown in the figure), which then initiates the campaign by executing the fuzzer on the instrumented SUT binary. Parallel to the campaign, FuzzTastic monitors the code regions covered by the generated inputs and stores the coverage data in report files (see Section 5.2.3 on page 68).

In **step** ❹, the saturation analyzer takes the identified bug-prone code regions and the currently covered regions, as provided by FuzzTastic, to determine the amount of covered, potentially vulnerable code (i.e., the score of our metric). If the analyzer detects a saturation of our metric within the specified saturation window and allowable deviation, it automatically terminates the campaign. Otherwise, it keeps the campaign running while performing this check again after a short timeout with the new coverage data.

6.4. Evaluation

In this section, we outline the setup for evaluating the cost-efficiency differences between our and the baseline stopping criteria for fuzzing campaigns, describe the obtained results, and discuss the main findings and insights.

6.4.1. Setup

Research Questions

This evaluation aims to show the superior cost-efficiency of our stopping criterion compared to the saturation of program crashes and regular code coverage. Therefore, we answer the following *research questions* through empirical evaluations:

RQ 3a: **Crash Saturation Comparison.** What is the tradeoff in terms of fuzzing time saved and security bugs missed when stopping fuzzing campaigns based on the saturation of vulnerability-oriented function coverage rather than the number of triggered program crashes?

RQ 3b: **Function Coverage Saturation Comparison.** What is the tradeoff in terms of fuzzing time saved and security bugs missed when stopping fuzzing campaigns based on the saturation of vulnerability-oriented function coverage rather than regular function coverage?

The above research questions compare the cost-efficiency of our stopping criterion with the two selected baseline criteria. As mentioned before, a more cost-efficiency stopping criterion would be very helpful for fuzzing in practice. It would allow stopping ineffective campaigns earlier, thus saving valuable resources, and running effective campaigns longer, thus increasing the chances of detecting bugs. Also, this evaluation provides insights into the accuracy of our vulnerability-oriented code coverage metric in reflecting fuzzing cost-efficiency compared to the baseline metrics, especially regular code coverage.

Table 6.1.: Subject programs employed for evaluating the cost-efficiency of our stopping criterion (as appeared in [170]).

Subject	Version	# Lines	# Functions	# Crashes	# Bugs
Gif2png	2.5.3	988	27	3,620	4
JasPer	1.900.0	17,385	720	9,853	77
Libpcap	1.9.0	12,076	497	595	7
LibTIFF	4.1.0	19,527	826	17,701	7
Libxml2	2.9.10	85,466	2,982	50,907	9
NM	2.29	68,667	2,126	58	4
Objdump	2.29	89,961	2,701	4,014	5
Size	2.29	68,115	2,101	77	6
Total		362,185	11,980	86,825	119

Dataset

For our evaluation, we use the FuzzTastic dataset described in Section 5.3 on page 69, which provides extensive code coverage and crash data from several widely used fuzzers executed on different real-world programs. Specifically, we include those programs from this dataset for which the fuzzers were able to detect at least one bug. Additionally, we fuzzed LibTIFF and Libxml2 from Magma [114] using the same software and hardware infrastructure used to generate the FuzzTastic dataset in order to increase the number of bugs in our evaluation. The final set of subject programs is shown in Table 6.1. Note that these programs are completely different from those used to train the ML vulnerability prediction models (see Table 4.2a on page 58).

Bug Deduplication

The fuzzing campaigns triggered a total of 86,825 program crashes. Since manually deduplicating this many crashes would be impractical, we used the approach described in Section 2.2.3 on page 23 to automatically deduplicate crashes into unique bugs. After manually reviewing a random sample of about 50 crash traces, we found that comparing the top-three stack frames (without the vulnerability type) provides the most accurate deduplication. Ultimately, applying this automatic deduplication approach yields 119 unique security bugs.

Selected Models and Cutoffs

As discussed in Section 4.2.4 on page 57, for optimal effectiveness, we recommend training, optimizing, and evaluating different ML vulnerability prediction models on older, vulnerable versions of the targeted software. The most effective model should then be selected for our metric and, thus, our stopping criterion. However, due to the limited number of vulnerabilities per program in our dataset, we trained the models on programs

Table 6.2.: Selected ML vulnerability prediction model and cutoff value for each subject program.

Subject	Model	Cutoff	Flagged Funcs.
Gif2png	Feedforward Neural Network	0.4	69.2%
JasPer	Random Decision Forest	0.2	72.6%
Libpcap	Stochastic Gradient Boosting	0.5	14.2%
LibTIFF	Random Decision Forest	0.7	13.4%
Libxml2	Stochastic Gradient Boosting	0.8	34.3%
NM	Random Decision Forest	0.8	8.5%
Objdump	Feedforward Neural Network	0.5	53.7%
Size	Random Decision Forest	0.8	8.6%

other than those used in this evaluation. To ensure a realistic evaluation, we then chose for each subject program the ML model and cutoff value that managed to identify at least 90% of the bug-triggering vulnerable functions with the fewest functions reachable[1] from the fuzz entry being flagged (see Table 6.2). We consider this a realistic detection rate threshold, as evidenced by related studies [148, 185, 187] as well as by our evaluation in Section 4.3.2 on page 59.

Metrics

Figure 6.5 on the next page illustrates our approach for evaluating the cost-efficiency differences, where the termination points of the compared stopping criteria are denoted as t_{S_1} and t_{S_2}, respectively. Specifically, we compute the ratio of fuzzing time saved and that of security bugs missed (not shown in the figure) between our and the selected baseline stopping criterion for a given fuzzing campaign. For the former, we divide Δt by the campaign length (24 hours) for a given saturation window δ and an allowable metric deviation ϵ (zero in this example). In contrast, for the latter, we divide the number of unique bugs found between t_{S_1} and t_{S_2} by the total number of bugs found in the respective subject program while generating the dataset.

We perform this evaluation across different fuzzers and subject programs to test whether the results generalize to a broader setup. Also, to examine the impact of different saturation configurations, we run the experiments with $\delta \in \{2\,\text{h}, 4\,\text{h}, 6\,\text{h}, 8\,\text{h}\}$ and $\epsilon \in \{0, 1, 2\}$, i.e., the number of functions or crashes (depending on the metric) allowed to be exercised within δ. Furthermore, we report the arithmetic means and standard deviations for both measures over 20 campaign repetitions to account for the randomness in fuzzing.

[1] We used the SVF static analysis framework [283] for the reachability analysis.

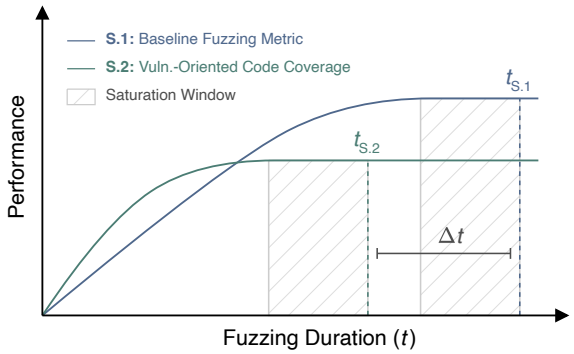


Figure 6.5.: Illustration of our approach for evaluating the cost-efficiency difference between our and a baseline stopping criterion (as appeared in [170]). The times t_{S_1} and t_{S_2} indicate the termination point of the respective criterion.

6.4.2. Results

Figures 6.6 and 6.8 on the facing page and on page 85 show the cost-efficiency differences between our stopping criterion and the baseline criteria for each subject-fuzzer pair with the different saturation configurations. For example, the LibTIFF-AFLFast cell in Figure 6.6 shows that with a saturation window of $\delta = 4\,\text{h}$ and an allowable deviation of $\epsilon = 0$ (crashes), our stopping criterion ends the 24-hour fuzzing campaigns, on average, 12 hours, i.e., 50 percentage points (pp), earlier than the crash saturation criterion while relatively missing not more than one out of seven bugs. Note that negative values indicate additional time spent or bugs found by our criterion.

Furthermore, Figures 6.7 and 6.9 on page 84 and on page 86 show the fuzzing time savings relative to the number of bugs missed of our stopping criterion with regard to the respective baseline criterion. Note that these two figures only consider the 1,020 (out of 1,280) campaigns in our dataset in which at least one bug could be triggered.

Crash Saturation Comparison

Figure 6.6 on the facing page shows that for the subjects Gif2png, JasPer, LibTIFF, and Libxml2, our stopping criterion terminates the campaigns, on average, 6 to 12 hours earlier than crash saturation (i.e., time savings of 25%–50%) while missing about 0.5 bugs (4%). For MOpt-AFL and MOpt-AFL++ executed on LibTIFF, savings of up to 18 hours (75%) could be achieved. In contrast, in Libpcap, NM, and Size, our criterion stops the campaigns 2.4 to 4.8 hours (10%–20%) later but often finds one additional bug for this tradeoff. Only in Objdump, when fuzzed with AFL, AFL++, AFLFast, and Honggfuzz, we observe that our criterion extends the time expenditures by about 6 to 12 hours (25%–50%).

Moreover, Figure 6.7 on page 84 shows that our stopping criterion is more cost-efficient than crash saturation for most (10 out of 12) of the studied ϵ and δ configurations. Across the different saturation configurations and ranges of missed bugs, our criterion saves, on average, 4.1 hours (17.2pp) of fuzzing time. Out of the 1,020 bug-finding fuzzing campaigns that are stopped with our criterion, 84.2% miss no bugs, 6.6% miss one bug, and 4.9% miss two or more bugs, on average, compared to crash saturation. Conversely,

Figure 6.6.: Cost-efficiency differences between our stopping criterion and crash saturation (as appeared in [170]).

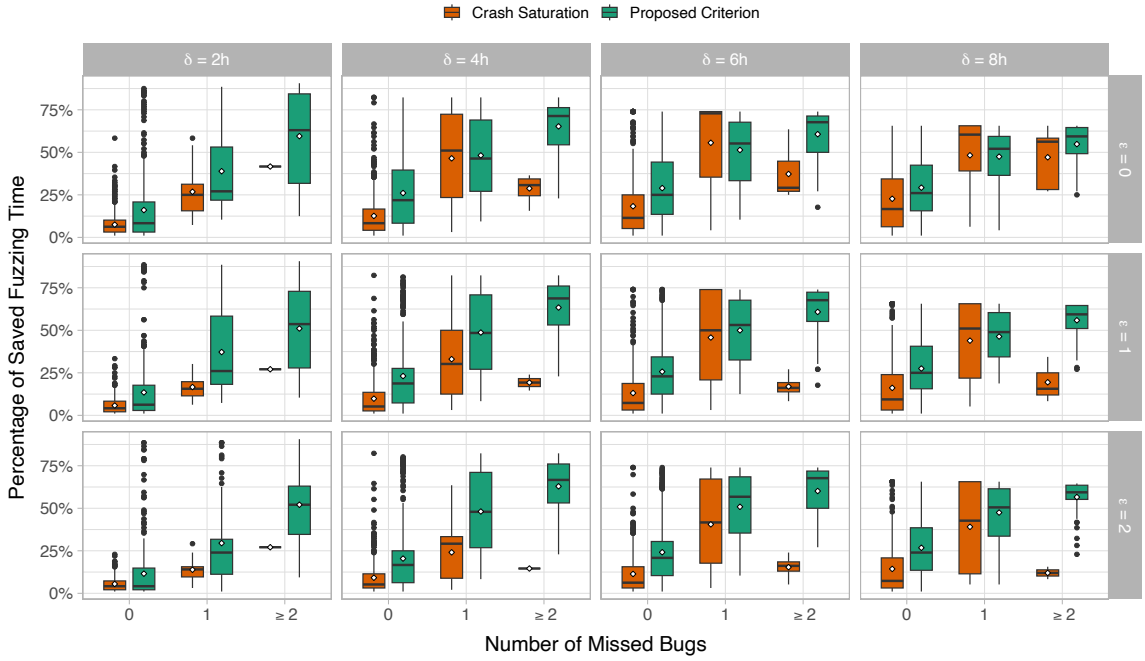


Figure 6.7.: Fuzzing time savings relative to the number of missed bugs of our stopping criterion with respect to crash saturation (as appeared in [170]). The diamonds (⋄) indicate the arithmetic means.

one additional bug is found in 4% of these campaigns and two or more in 0.3% when using our criterion.

> **Summary (RQ 3a)**
>
> Our evaluation shows that our vulnerability-oriented stopping criterion is more cost-efficient than crash saturation. On average, it stops 24-hour fuzzing campaigns 4.1 hours (17.2pp) earlier across the studied ϵ and δ configurations. Thereby, our criterion misses bugs in 11.5% of the 1,020 bug-finding campaigns but also uncovers bugs that crash saturation misses in 4.3% of these campaigns.

Function Coverage Saturation Comparison

Figure 6.8 on the facing page shows that for many subject-fuzzer pairs, our criterion stops the campaigns, on average, 4.8 to 9.6 hours earlier than function coverage saturation (i.e., time savings of 20%–40%) while missing only 0.2 bugs (2.7%). For some pairs, such as Size-AFLFast, our criterion even saves up to 12 hours (50%) of fuzzing time. However, no significant savings could be achieved in JasPer. This is because JasPer, with 77 known security bugs (see Table 6.1 on page 80), contains many vulnerable functions that are also identified as such by the ML vulnerability prediction model so that the campaigns continuously increase the coverage of these bug-prone functions. This causes our criterion not to end these campaigns earlier.

Figure 6.8.: Cost-efficiency differences between our stopping criterion and function coverage saturation (as appeared in [170]).

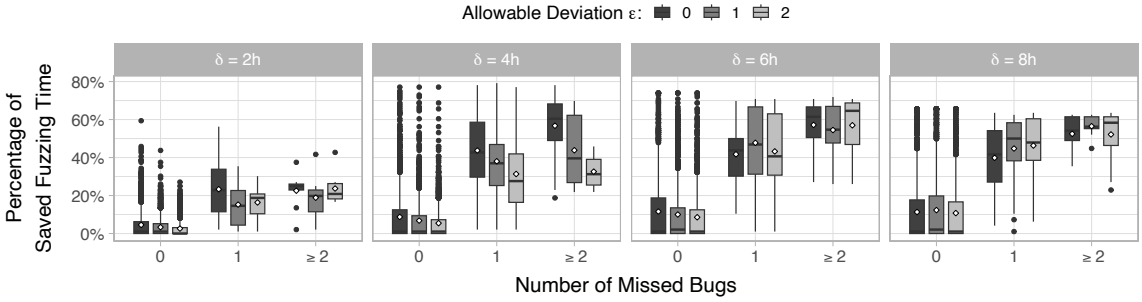


Figure 6.9.: Fuzzing time savings relative to the number of missed bugs of our stopping criterion with respect to function coverage saturation (as appeared in [170]). The diamonds (⋄) indicate the arithmetic means.

Moreover, Figure 6.9 shows that our stopping criterion reduces the fuzzing time, on average, by 1.7 hours (7%) in campaigns where no bugs are missed across the studied ϵ and δ configurations. In campaigns where bugs are missed, on average, 9.6 hours (40.1%) could be saved. Out of the 1,020 bug-finding campaigns stopped with our criterion, 94.3% miss no bugs, 4.6% miss one bug, and 1.1% miss two or more bugs, on average, compared to function coverage saturation.

Summary (RQ 3b)

Our evaluation shows that our vulnerability-oriented stopping criterion is more cost-efficient than function coverage saturation. On average, it stops 24-hour fuzzing campaigns 4.8 to 9.6 (20pp–40pp) hours earlier while missing only 0.2 out of 7.4 bugs across the studied ϵ and δ configurations. Thereby, our criterion misses bugs in 5.7% of the 1,020 bug-finding campaigns but saves an average of 9.6 hours in these campaigns.

6.4.3. Discussion

Although crash saturation proved effective as a stopping criterion for some of the fuzzing campaigns in our evaluation, we argue that coverage-based criteria are more reliable. Böhme and Falk [22] empirically found that each new security bug discovery requires exponentially more time or hardware resources. Consequently, with a fixed number of machines, bugs will be discovered later and later. Thus, the crash saturation criterion becomes *impractical* at some point because campaigns are likely to be terminated far before any bugs are detected. In contrast, coverage-based stopping criteria usually capture the progress of a campaign more accurately and are, therefore, the better choice.

Our evaluation shows that our vulnerability-oriented stopping criterion can terminate fuzzing campaigns *significantly earlier* than the saturation of program crashes or regular code coverage, with little risk of missing bugs. This also demonstrates the improved accuracy of our metric in reflecting fuzzing cost-efficiency. For example, using our stopping criterion in a setting where the latest software builds are fuzzed overnight for 10 hours after each of the five workdays, a 50% time reduction (possible with our criterion)

saves 1,200 fuzzing hours in one year, 2,400 hours in two years, 6,000 hours (about 250 days) in five years, and so on.

A limitation of our criterion is that it may stop campaigns too early, which can lead to missed bugs. This can happen if the campaign stagnates on the identified potentially vulnerable code region for too long before triggering a bug in the same or another vulnerable region. To mitigate this, we suggest using a longer saturation time window or running additional campaigns once or twice a month with a fixed time budget (e.g., 48 hours), which still allows for large resource savings.

6.5. Threats to Validity

To minimize threats to *external validity*, we conducted our tradeoff evaluation across eight different subject programs containing a total of 119 security bugs and eight different fuzzers. This diverse setup should sufficiently ensure that our results are not limited to a particular program or fuzzer.

Regarding *internal validity*, the large number of program crashes in our dataset forced us to automate the deduplication of crashes into unique bugs. We used a well-established heuristic from the fuzzing domain, assuming that crashes with similar stack frames share the same underlying vulnerability. To reduce the risk of incorrect approximations, we determined the number of frames considered during deduplication by manually analyzing a randomly sampled subset of stack traces from crashing executions.

Furthermore, to account for the randomness in fuzzing, we followed the guidelines of Klees et al. [144] in our evaluation and reported the arithmetic means over 20 campaign repetitions.

6.6. Summary

In this chapter, we introduced our criterion to stop fuzzing campaigns based on the saturation of our vulnerability-oriented code coverage metric, i.e., the covered, potentially vulnerable code. To statically identify the bug-prone code before fuzzing, we use the best ML vulnerability prediction model(s) from Chapter 4 on page 49. In a large-scale evaluation, we showed that our stopping criterion is more cost-efficient than the saturation of program crashes and regular code coverage. Specifically, our criterion could achieve significant fuzzing time savings with a low risk of missing bugs compared to the baseline criteria, which is particularly valuable in practice since fuzzing is very resource-intensive. Ultimately, this demonstrates the superior accuracy of our metric in reflecting fuzzing (cost-)efficiency compared to the two baseline metrics, especially regular code coverage. In the next chapter, we integrate our metric into a fuzzer to also demonstrate improved accuracy in reflecting fuzzing effectiveness.

7. Vulnerability-Guided Fuzzing

This chapter presents a novel fuzzing approach that integrates our vulnerability-oriented coverage metric as the main test-optimization criterion into a directed fuzzer to target potentially vulnerable code. By comparing the effectiveness of our approach in detecting security bugs with coverage-based and existing vulnerability-guided fuzzing approaches, we demonstrate the superior accuracy of our metric in reflecting fuzzing effectiveness. Parts of this chapter have appeared in a submission [171], first-authored by the author of this thesis.

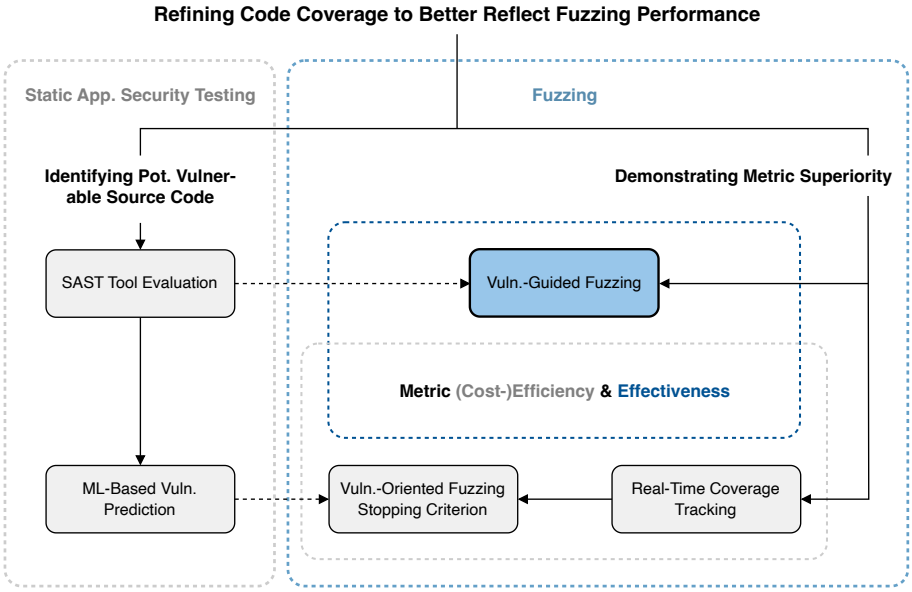


Figure 7.1.: Big Picture (Contribution 5: Vulnerability-Guided Fuzzing).

7.1. Introduction

Context. As shown in Figure 7.1, this chapter belongs to the part of this dissertation where we demonstrate the improved accuracy of our vulnerability-oriented code coverage metric in reflecting fuzzing effectiveness over regular code coverage and other existing vulnerability-oriented metrics. For this, we integrate our metric into a fuzzer to optimize the test generation and then compare its effectiveness in detecting security bugs against fuzzers using the respective baseline metrics. Thus, this chapter not only examines the effectiveness of our metric but also introduces a novel fuzzing approach.

State-of-Practice and Problem. Current fuzzing approaches primarily aim to maximize overall code coverage [25, 53, 86, 99, 157, 175, 235, 240, 326], with newer vulnerability-guided approaches [306] targeting specific parts of the SUT such as recently modified [336], complex [73], memory-accessing [305], or sanitizer-instrumented code [222, 265, 334]. However, due to the limitations discussed in Section 1.1.3 on page 5, coverage-based fuzzing has been shown to be prohibitively expensive, requiring an *exponential* increase in time or hardware resources to achieve only a linear increase in bug discovery [22]. Furthermore, current vulnerability-guided fuzzing is either limited to incremental fuzzing of code regressions provided by the VCS commit history or faces similar problems as coverage-based fuzzing because it relies on *imprecise* vulnerability predictors. As we will show in the next section, these predictors flag a lot of code as potentially vulnerable and, therefore, require again extensive code coverage.

Solution Approach. To address this research gap, we integrate our vulnerability-oriented code coverage metric as a *test-optimization criterion* into a directed fuzzer [24], called SASTFuzz, thereby leveraging SAST tools to identify the potentially vulnerable code that should be targeted during fuzzing. As part of this approach, we introduce a target scheduling mechanism that *dynamically* disables likely unreachable and false-positive SAST tool findings used as fuzzing targets from the distance calculations performed during directed fuzzing (see Paragraph "Coverage of Specific Targets" on page 21). This allows the fuzzing campaign to pivot more quickly to other, more likely vulnerable targets, thus mitigating the impact of such targets on the fuzzing performance. We also dynamically switch between vulnerability-guided and coverage-based fuzzing, depending on the reachability of the targets, to account for vulnerable code missed by the SAST tools (i.e., false negatives).

Contributions. Overall, we contribute a vulnerability-guided fuzzing approach that optimizes the test generation according to our metric. Our results show that our dynamic target scheduling typically leads to a higher target coverage than fuzzing with static targets. Moreover, compared to coverage-based fuzzing, our fuzzing approach detects 10 to 14 additional previously unknown security bugs with 3 to 4.5 times more different ones. Compared to the best vulnerability-guided baseline approaches, it detects 5 to 12 additional bugs, with 1.6 to 3 times more different ones. Against the least effective vulnerability-guided baseline, it detects all the bugs found by that approach plus 63 additional bugs. Thus, these results demonstrate the improved accuracy of our metric in reflecting fuzzing effectiveness. Additionally, our approach finds the bugs in most subject programs, on average, up to 10.5 and 5.8 hours earlier during 48-hour campaigns than the fastest coverage-based and vulnerability-guided fuzzing baselines, respectively.

7.2. Motivation

Unlike coverage-based fuzzing, vulnerability-guided approaches focus on testing specific code locations that are more likely to be vulnerable than others. However, as mentioned above, current vulnerability-guided fuzzing relies on imprecise vulnerability heuristics

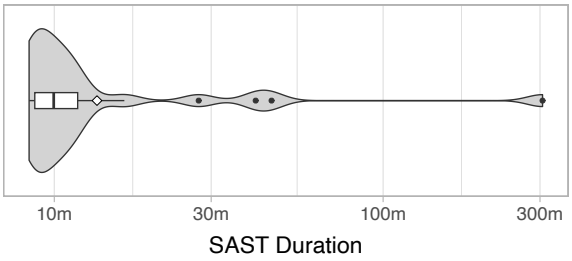


Figure 7.2.: Distribution of static application security testing (SAST) durations across 24 different real-world subject programs when analyzed in parallel by five state-of-the-art SAST tools (as appeared in [171]). The diamond (⋄) indicates the arithmetic mean.

that often produce numerous fuzzing targets, most of which are false positives, which can lead to ineffective fuzzing. Therefore, we suggest using SAST tools to accurately identify such targets, presenting hereafter concrete examples that demonstrate their low runtime overhead and high effectiveness in finding bug-prone code.

Runtime Overhead. Figure 7.2 shows the SAST durations of five state-of-the-art SAST tools executed[1] in parallel across the 24 real-world programs described in Sections 7.4.1 and 7.5.1 on page 100 and on page 103. Specifically, we employ the freely available analyzers Flawfinder, Infer, CodeQL, Clang Static Analyzer, and Semgrep (see Section "SAST Tools" on page 93), which all implement advanced static analysis techniques.

The figure shows that most analysis runs complete in less than 15 minutes, although Pocketlang, Libxml2, and libsodium take between 27 and 45 minutes. The largest outlier is OpenSSL, one of our largest subject programs, where Infer takes about 5 hours to process the 166,814 LoC[2]—an unusually long time considering that Infer requires significantly less time for other subjects with similar LoC. Overall, the median duration is less than 10 minutes, thus introducing only a *small* time overhead for our SAST-guided fuzzing approach.

Bug-Finding Effectiveness. Table 7.1 on the next page shows the effectiveness[3] of the five selected static application security testing tools when used in combination versus that of AddressSanitizer in identifying vulnerable functions in five of the subject programs. A vulnerable function is thereby considered detected if at least one of its lines is flagged, regardless of the reported vulnerability type.

In the subjects Libpng, Libsndfile, and Libxml2, the SAST tools reliably identify the vulnerable functions with detection rates ranging from 0.78 to 0.93, slightly lower than AddressSanitizer's rates of 0.86 to 1.00. However, the analyzers only flag 25.6% to

[1] We limited the analyzers' hardware access to 24 cores and 64 GB RAM to measure the overhead under common hardware conditions.

[2] We recommend a full initial codebase scan for larger programs, followed by incremental SAST runs focusing on code changes [117].

[3] The precision values must be interpreted with a grain of salt, as it is uncertain whether there are still undiscovered vulnerabilities in the subject programs.

Table 7.1.: Comparison of the effectiveness of the five selected static application security testing (SAST) tools (used in combination) versus AddressSanitizer (ASan) in identifying vulnerable functions (as appeared in [171]).

	Flagged Funcs.		Recall		Precision	
Subject	SAST	ASan	SAST	ASan	SAST	ASan
Libpng	25.6%	81.4%	0.778	1.000	0.064	0.028
Libsndfile	36.4%	73.0%	0.929	0.857	0.029	0.013
LibTIFF	23.8%	78.1%	0.385	1.000	0.025	0.036
Libxml2	38.9%	80.3%	0.857	1.000	0.011	0.009
OpenSSL	21.8%	58.3%	0.682	1.000	0.005	0.004
Mean	29.3%	74.2%	0.726	0.971	0.027	0.018

38.9% of the functions, which is significantly less than the 73% to 81.4% of the functions flagged by AddressSanitizer. In OpenSSL, the SAST detection rate drops slightly to 0.68 while flagging 21.8% of the functions. Although AddressSanitizer identifies all vulnerable functions, it comes at the cost of 58.3% functions flagged. Regarding LibTIFF, the analyzers largely underperform compared to the sanitizer—they miss more than half of the vulnerable functions, which is at odds with their otherwise strong performance. However, across all five subject programs, the SAST tools, when used in combination, have an average detection rate of 0.73 while flagging 29.3% of the functions in the SUT[4]. Although less effective than AddressSanitizer, they flag 2.5 times fewer functions and even fewer compared to vulnerability heuristics that focus on all memory modifications. Ultimately, this shows that SAST tools can *accurately* and *fairly reliably* identify potentially vulnerable functions.

To mitigate the impact of the many false positives (average precision of 0.03), we systematically deactivate fuzzing targets. Furthermore, to reduce the risk of false negatives, we strategically switch between directed and coverage-based fuzzing. Based on these results and the two mechanisms to mitigate the SAST limitations (discussed in Section 7.3.2 on page 94), we hypothesize that our fuzzing approach leads to more effective fuzzing than coverage-based fuzzing and existing vulnerability-guided approaches.

7.3. Vulnerability-Guided Fuzzing Approach

In this section, we describe our vulnerability-guided fuzzing approach, implemented in our SASTFuzz fuzzer, which builds upon WindRanger [74]. While SASTFuzz inherits WindRanger 's mutation operators and target execution feedback, it integrates a novel SAST-based target acquisition and directed fuzzing approach that supports dynamic targets. Below, we first outline the SAST tools and methods used to identify, group, and

[4]These results are consistent with the best static analyzer combination from our SAST tool evaluation in Figure 3.10 on page 43, but this time without relying on the dominant proprietary analyzer CommSCA, using only free tools.

weight the fuzzing target locations in the SUT's source code. After that, we delve into directed fuzzing, explaining the modified distance measure to guide fuzzing towards these targets, as well as the mechanism for dynamic target and fuzzing mode scheduling.

7.3.1. SAST-Based Target Acquisition

SAST Tools

We employ several *advanced* and *freely available* SAST tools to (i) maximize the effectiveness of identifying potentially vulnerable, (ii) make our approach accessible without the need for a costly proprietary tool license, (iii) support multiple vulnerability types, and (iv) determine the likelihood of code being vulnerable by analyzer consensus. Specifically, we use the latest versions available at the time of this study of the four most effective free analyzers from our SAST tool evaluation in Section 3.4 on page 40 and related studies [94, 102]: Flawfinder (version 2.0.19) [310], Infer (1.1.0) [188], CodeQL (2.12.0) [95], and Clang Static Analyzer (12.0.0) [237] (see the tool descriptions in Section 3.2.1 on page 30). In addition, we include Semgrep (1.24.0) [262], which has gained great popularity as evidenced by its 10.8k GitHub stars [28]. Semgrep implements a static analysis technique that is based on abstract syntax tree (AST) pattern matching and lightweight data-flow analysis, where security checks are specified via configuration files.

SAST Finding Grouping

To remove redundant fuzzing targets, we group the SAST tool findings at the BB level. Specifically, all flagged code lines within the same BB are considered a single target location (i.e., the starting line number of that BB), which is then referred to as target basic block (TBB). Note that this is a valid reduction because all (flagged) lines in the TBB are executed sequentially once it is reached.

Definition 7.1 (Target Basic Blocks)**.** We define the set of target basic blocks B_t as those basic blocks in the SUT that contain at least one code line flagged as potentially vulnerable by the employed SAST tools.

Unlike most of the related works [24, 74, 162], which group the target locations during program instrumentation, we initiate this process earlier in the static analysis phase. This way, we *speed up* the distance calculations and thus the instrumentation, which is currently one of the main bottlenecks of directed fuzzing [174].

Target Basic Block Weighting

After grouping the SAST tool findings into TBBs, we assign weights to them based on their likelihood of being vulnerable. The idea is that a higher weight, hereafter called *vulnerability score*, correlates with a higher probability that the respective TBB contains a vulnerability. This weighting then allows for (i) an improved target prioritization by fuzzing TBBs with higher vulnerability scores first and (ii) a systematic resource allocation by devoting more fuzzing time to targets with higher scores.

Code Granularity. We use *function-level* SAST information to weight the TBBs. This is because the available ground-truth data that we need to validate our weighting approach in Section 7.4 on page 99 only includes the code locations where vulnerabilities originate, not where they manifest, which are the locations that are typically flagged by SAST tools, as exemplified in Listing 3.1 on page 36. However, since both locations usually occur within the same function (see Table 3.3 on page 38), we consider the function level a valid granularity for approximating SAST tool bug finding and thus for validating our weighting. As a result, all TBBs within the same function are assigned the same vulnerability score.

SAST-Based Vulnerability Heuristics. Inspired by the tool-based features used to build the ML vulnerability prediction models in Section 4.2.1 on page 51, we use the following *vulnerability heuristics* in order to determine the vulnerability score of a TBB.

Definition 7.2 (SAST Tool Consensus)**.** For a target basic block $b_t \in B_t$ in function $f \in F$, we define the SAST tool consensus heuristic as the ratio of analyzers that flagged f as potentially vulnerable, i.e., the number of SAST tools that flagged one or more lines in f divided by the number of all tools that analyzed f.

Our first heuristic in Definition 7.2 is based on the *majority voting* principle, assuming that a consensus among several static analyzers about flagging a function increases the likelihood that the function is vulnerable.

Definition 7.3 (SAST Flag Density)**.** For a target basic block $b_t \in B_t$ in function $f \in F$, we define the SAST flag density heuristic as the ratio of lines in f that are flagged as potentially vulnerable by the analyzer(s), i.e., the number of flagged lines in f divided by the number of all lines in f.

Our second heuristic in Definition 7.3 is based on the assumption that a higher number of flagged lines in a function indicates a higher density of critical statements, which, in turn, increases the likelihood that the function is vulnerable.

7.3.2. Directed Fuzzing with Dynamic Targets

Distance Measure

Unlike coverage-based fuzzing, which aims to cover all code uniformly, directed fuzzing aims to reach specific code locations. For this, it tries to minimize the distance between the execution traces of the generated fuzz inputs and the target locations, as described in Paragraph "Coverage of Specific Targets" on page 21. Our directed fuzzer is thereby built on top of WindRanger [74], which improves the "directedness" of AFLGo [24] in two ways.

First, WindRanger uses data-flow information to account for the complexity of conditions leading to targets. Specifically, it mutates fuzz inputs that navigate less complex conditions but follow a longer execution trace more often than those with shorter traces (to the same targets) but more complicated conditions.

Second, WindRanger refines the distance measure using detailed control-flow information. Instead of calculating the distance from all BBs on the execution trace to the TBBs, it

calculates the distance from only a subset of so-called deviation basic blocks (DBBs) that are statically identified before fuzzing. A DBB thereby acts as a *critical waypoint* where an execution trace might deviate from reaching any TBBs, thus providing precise guidance for the fuzzing campaign.

Definition 7.4 (Deviation Basic Blocks)**.** For a given set of target basic blocks B_t, Du et al. [74] define the set of deviation basic blocks B_d as the first basic blocks in the potential program execution traces where at least one sub-trace deviates from reaching any $b_t \in B_t$.

Formally, the refined distance measure can be described by the function `distance` : $B_d \times B_t \to \mathbb{N} \cup \{\text{-1}\}$ as follows:

$$\texttt{distance}(b_d, b_t) := \begin{cases} \text{\# Edges of the} & \text{if } b_t \text{ is statically reachable from } b_d, \\ \text{shortest iCFG path,} & \\ \text{-1,} & \text{otherwise.} \end{cases} \tag{7.1}$$

The measure in Equation (7.1) more accurately reflects the ability of a fuzz input to reach a TBB, resulting in a better input prioritization. This means that inputs with longer distances can still be prioritized as long as they progress towards the TBBs.

Dynamic Target Scheduling

To mitigate the impact of likely unreachable and false-positive targets on the fuzzing performance, we (i) *temporarily* suspend targets that remain unreached for increasingly longer periods and (ii) *permanently* deactivate targets once the fuzz inputs have executed them "sufficiently often". For this, each TBB $b_t \in B_t$ is during fuzzing either *activated* (initial state), *paused* (temporarily suspended), or *completed* (permanently deactivated), with the respective TBB state being stored in the variable b_t.state. Thereby, only the activated TBBs are included in the distance calculations.

To temporarily suspend targets, we use a *dynamic scheduling mechanism* that toggles unreached TBBs based on b_t.{ivalSkips, ivalSkipsPrev} (discussed in detail below). To permanently deactivate targets, we use a *hit-count threshold* that must be exceeded, which is derived from the TBB's (relative) vulnerability score (b_t.score) and its input hit count (b_t.inputHits).

Target Deactivation and Scheduling. Algorithm 7.1 on the following page outlines our scheduling strategy, which is executed at *intervals* during fuzzing to (re-)assess the state of the uncompleted TBBs. It takes as parameters the initially specified TBBs (B_t) and the number of fuzz inputs generated during the last interval (*numIvalInputs*). The algorithm itself then consists of two parts: the first part deactivates sufficiently fuzzed TBBs (lines 3–5), and the second part schedules potentially unreachable TBBs (lines 7–21).

PART 1: While iterating over all targets b_t that are either activated or paused, line 3 calls the function `calcHitCountThreshold` from Algorithm 7.2 on the next page to determine the hit-count threshold (i.e., the number of input hits required before b_t is marked as completed). In this function, line 2 calculates the vulnerability score of b_t relative to all

Algorithm 7.1: Dynamic target BB scheduling (as appeared in [171]).

```
   Data: B_t, numIvalInputs ∈ ℕ
   Result: Updated B_t
1  foreach b_t ∈ B_t do
2      if b_t.state ∈ {active, paused} then
3          hitCountThreshold ← calcHitCountThreshold(b_t, B_t, numIvalInputs);
4          if (hitCountThreshold − b_t.inputHits) ≤ 0 then
5              b_t.state ← completed;
6          else
7              if coveredInPrevInterval(b_t) then
8                  b_t.state ← active;
9                  b_t.ivalSkips ← 0;
10                 b_t.ivalSkipsPrev ← 1;
11             else
12                 if b_t.ivalSkips = 0 then
13                     b_t.state ← paused;
14                     b_t.ivalSkips ← b_t.ivalSkipsPrev;
15                     b_t.ivalSkipsPrev ← b_t.ivalSkipsPrev + 1;
16                 else if (b_t.ivalSkips − 1) = 0 then
17                     b_t.state ← active;
18                     b_t.ivalSkips ← 0;
19                 else
20                     b_t.ivalSkips ← b_t.ivalSkips − 1;
21                 end
22             end
23         end
24     end
25 end
```

Algorithm 7.2: Computation of the hit-count threshold (as appeared in [171]).

```
1  Function calcHitCountThreshold(targetBB ∈ B_t, B_t, numIvalInputs ∈ ℕ)
2      relTargetBBScore ← targetBB.score / sum({b_t.score | b_t ∈ B_t ∧ b_t.state ≠ completed});
3      return numIvalInputs * relTargetBBScore;
```

active or paused TBBs. Line 3 then sets the threshold value by multiplying the score with the total number of inputs generated in the last interval. Consequently, a higher vulnerability score necessitates more input hits to complete the TBB, resulting in more thorough fuzzing.

Now, returning to Algorithm 7.1, if b_t has been executed sufficiently often, as determined by the if-condition in line 4, b_t will be deactivated by transitioning it from a paused or active state to the completed state. Formally, this transition is represented in our scheduling as $\Phi(b_t) = \langle \ldots, a \text{ or } p \rightarrow c, \ldots, c \rangle$, where $\Phi(\cdot)$ denotes the states of a TBB in the different intervals and "$\rightarrow$" the transition from one state to another.

PART 2: If b_t remains insufficiently fuzzed but was covered at least once in the previous interval (indicated by the function `coveredInPrevInterval`), it is directly activated for the next interval in line 8, i.e., $\Phi(b_t) = \langle \ldots, a \text{ or } p \rightarrow a, \ldots \rangle$. Also, the variables involved in pausing TBBs are reset due to the current reachability of b_t. However, if b_t remains uncovered for several intervals or the entire campaign, the scheduling in lines 12–21 follows

Algorithm 7.3: Distance computation to the target BBs (as appeared in [171]).

```
1  Function calcTargetBBsDistance(b_d ∈ B_d, B_t)
2      distance, n ← 0;
3      foreach b_t ∈ B_t do
4          targetDist ← distance(b_d, b_t);
5          if b_t.state = active and targetDist > 0 then
6              distance ← distance + 1/targetDist;
7              n ← n + 1;
8          end
9      end
10     if n > 0 then return n/distance;
11     else return −1;
```

the pattern: $\Phi(b_t) = \langle a, p, a, p, p, a, p, p, p, a, \ldots \rangle$—it alternately activates and pauses b_t, thereby incrementing the number of pausing intervals. This way, we account for likely unreachable TBBs by gradually reducing their impact on the distance calculation until they are reached by chance or the campaign ends.

Target Distance Calculation and Prioritization. We use the function in Algorithm 7.3 to compute the harmonic mean distance from a given DBB to all active TBBs. These average distances are stored in a *lookup table* for all DBBs, which is then used to calculate the distance for an executed fuzz input as the arithmetic mean over the stored distances of the covered DBBs. To keep the computational overhead low, this table is *only* updated with the average distances to the current set of activated TBBs after intervals that change the state of at least one TBB. Additionally, after each distance update, we reorder the fuzzer queue to mutate inputs with shorter distances to the updated target set first, thus increasing the chances of reaching TBBs. Thereby, we also prioritize TBBs with higher vulnerability scores.

Increasing Interval Length

As mentioned above, we (re-)assess the state of the uncompleted TBBs according to Algorithm 7.1 on the facing page at certain intervals during the fuzzing campaign. Starting with a *user-defined* initial interval length, the intervals increase *logarithmically* over the course of the campaign, as shown in Figure 7.3 on the next page. In this example, the campaign is started with an initial interval of 480 minutes (8 hours). After the first interval has elapsed, the active TBBs for the next interval are determined, followed by entering the second interval, which is 487.9 minutes (a slight logarithmic increase). The next assessment then occurs after the first plus second interval, i.e., after 967.9 minutes (16.1 hours) of fuzzing, with the third interval being increased to 490.2 minutes. This procedure continues until the end of the campaign.

As a result of this interval increase, progressively more fuzzing time is spent on TBBs that are difficult to reach, such as those deeply located in the SUT. Compared to a linear increase, this approach maintains reasonable assessment intervals while allocating more time to difficult TBBs. As deeper targets often correspond to more intricate vulnerabilities [12],

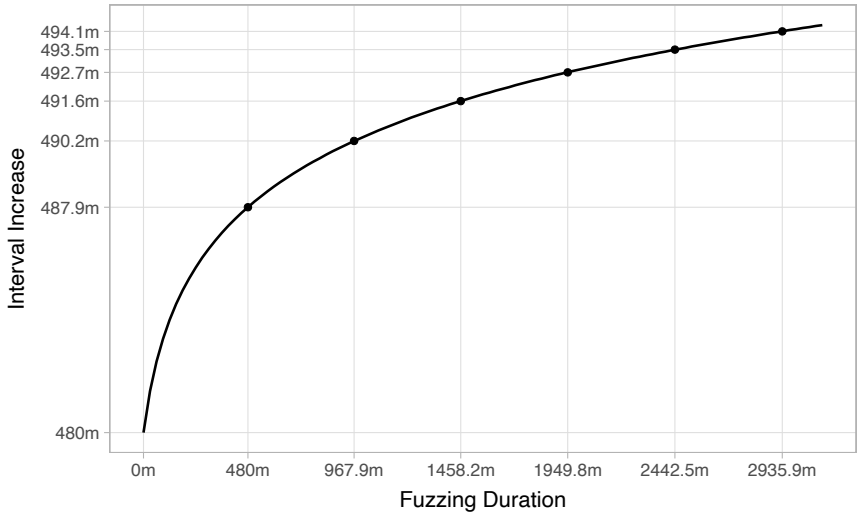


Figure 7.3.: Visualization of the logarithmic interval increases throughout the fuzzing campaign, starting with an initial interval length of 480 minutes (8 hours). Both axes display the durations in minutes, and the dots indicate transitions from one interval to the next.

the extended intervals ensure more input hits to mark such targets as completed, thereby increasing the likelihood of triggering complex vulnerabilities.

Obviously, the length of the (initial) interval has a significant impact on fuzzing performance. For instance, longer intervals lead to more thorough fuzzing of individual TBBs but may cover fewer targets overall. In contrast, shorter intervals are likely to cover more targets but run into the risk of missing bugs due to premature TBB deactivation. In Section 7.4 on the facing page, we examine this tradeoff in detail to determine an optimal initial interval for our purposes.

Dynamic Fuzzing Mode Switching

As already explained in the background chapter, (traditional) directed greybox fuzzing, as implemented in AFLGo, uses the initial exploration phase to maximize code coverage in order to expand the seed corpus and further avoid local optima. The subsequent exploitation phase is then in charge of directing fuzzing to the specified TBBs [24].

WindRanger, the technical foundation of SASTFuzz, enhances AFLGo by introducing a *dynamic switching strategy* based on the execution frequency of the DBBs [74]. Building upon this, SASTFuzz switches to the exploration mode for the duration of the next interval when all TBBs are paused or completed, allowing for the discovery of new or vulnerable code missed by the SAST tools. The process returns to an exploitation interval once a new TBB is reached or a previously paused one resumes activity. In case all TBBs are completed, the targets with a vulnerability score of 0.5 or higher are reactivated, focusing again on the more critical TBBs.

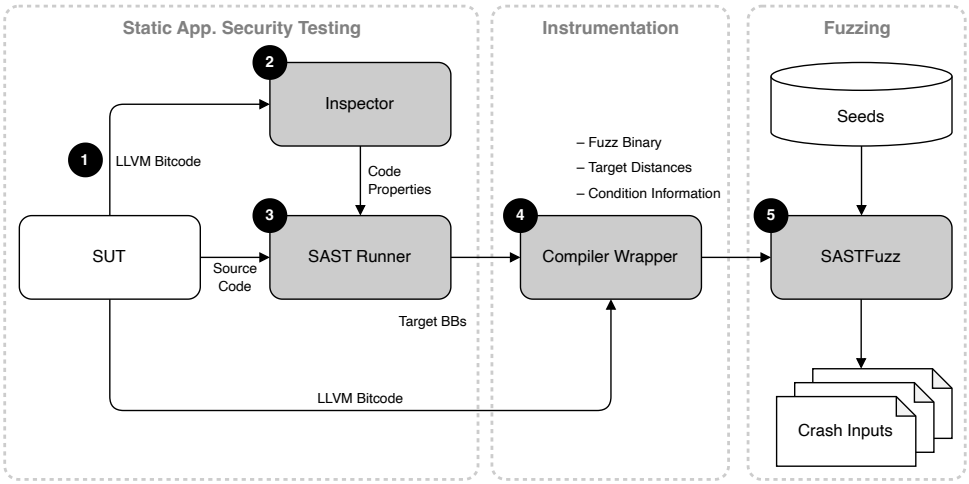


Figure 7.4.: Big picture showing the high-level system architecture of SASTFuzz (as appeared in [171]), which includes identifying target basic blocks with the static application security testing (SAST) tools, instrumenting the system under test (SUT) to collect the relevant coverage information, and, finally, fuzzing the resulting binary.

7.3.3. Application

Figure 7.4 shows the system architecture and workflow of SASTFuzz, which comprises three phases: static code analysis, program instrumentation, and fuzzing. Running SASTFuzz then involves the following steps:

In **step** 1, the LLVM bitcode file [153] of the SUT must be extracted.

In **step** 2, SASTFuzz's inspector component uses the extracted bitcode file to perform a whole-program analysis, identifying code properties such as the line ranges of all functions and BBs contained in the SUT.

In **step** 3, SASTFuzz's SAST runner component executes multiple static analyzers in parallel on the SUT to detect potentially vulnerable code. Thereby, it utilizes the code properties extracted in the previous step to group the SAST tool findings into TBBs and then assigns the corresponding vulnerability scores to them.

In **step** 4, the workflow transitions from static code analysis to program instrumentation, with the bitcode file finally being compiled into a fuzzable binary. During this process, SASTFuzz's compiler wrapper applies the necessary coverage instrumentation, identifies the relevant DBBs, and outputs the deviation-to-target BB distances along with their program path complexity information.

In **step** 5, SASTFuzz starts fuzzing the instrumented target program, thereby trying to reach the TBBs.

7.4. Preliminary Study

In this section, we empirically investigate (i) the validity of our SAST-based vulnerability heuristic for assigning the TBB vulnerability scores and (ii) the impact of different initial

interval lengths in our dynamic target scheduling on the fuzzing effectiveness. The gained insights will then guide the *configuration* of SASTFuzz for its comparative evaluation against the baseline fuzzers in Section 7.5 on page 103.

7.4.1. Setup

Subject Programs

For this preliminary study, we randomly select five programs, Libpng, Libsndfile, LibTIFF, Libxml2, and OpenSSL, from the Magma dataset [114], which we have already used in some of the evaluations in the previous chapters. Together, they contain a total of 97 vulnerable functions, with the specifics of Libpng, LibTIFF, Libxml2, and OpenSSL shown in Table 4.2 on page 58. In addition, we include Libsndfile (version 1.1.0), which consists of 1,224 functions, with 14 being vulnerable.

Seeds, Duration and Repetitions

We use for all fuzzing experiments a non-empty seed corpus that is provided by Magma. Moreover, we run each fuzzing campaign for 24 hours and repeat it five times to reduce any bias due to the randomness in fuzzing, which we consider sufficient to see the impact of the studied parameters.

Infrastructure

We performed all experiments in this preliminary study on a machine with an AMD EPYC^{TM} 7742 CPU containing 128 logical cores that run at 3.4 GHz, with access to 995 GB RAM and GNU/Linux Ubuntu 20.04 (64-bit) as the operating system.

7.4.2. Results

SAST-Based Vulnerability Heuristic Validity

To validate the two proposed vulnerability heuristics for assigning the TBB vulnerability scores, we calculate the *conditional probability* that a function is vulnerable given different levels of SAST tool consensus (see Definition 7.2 on page 94) and SAST flag density (see Definition 7.3 on page 94). We hypothesize that a higher heuristic score *correlates* with an increased probability of finding a vulnerable function, as measured by the Pearson correlation coefficient [259].

Regarding the SAST tool consensus heuristic, Figure 7.5 on the facing page shows a positive trend line for almost all of the five subject programs, thus confirming the heuristic's validity. In particular, Libsndfile and Libxml2 show a strong positive correlation ($\rho = 0.79$). In the other subjects, there is a strong positive trend up to $N = 3, 4$, after which the correlation decreases due to zero probabilities at the next higher values of N. This results in weak to moderate correlations in OpenSSL ($\rho = 0.16$) and Libpng ($\rho = 0.26$), and a negligible one in LibTIFF ($\rho = 0.08$), despite strong positive trends up to $N = 3$.

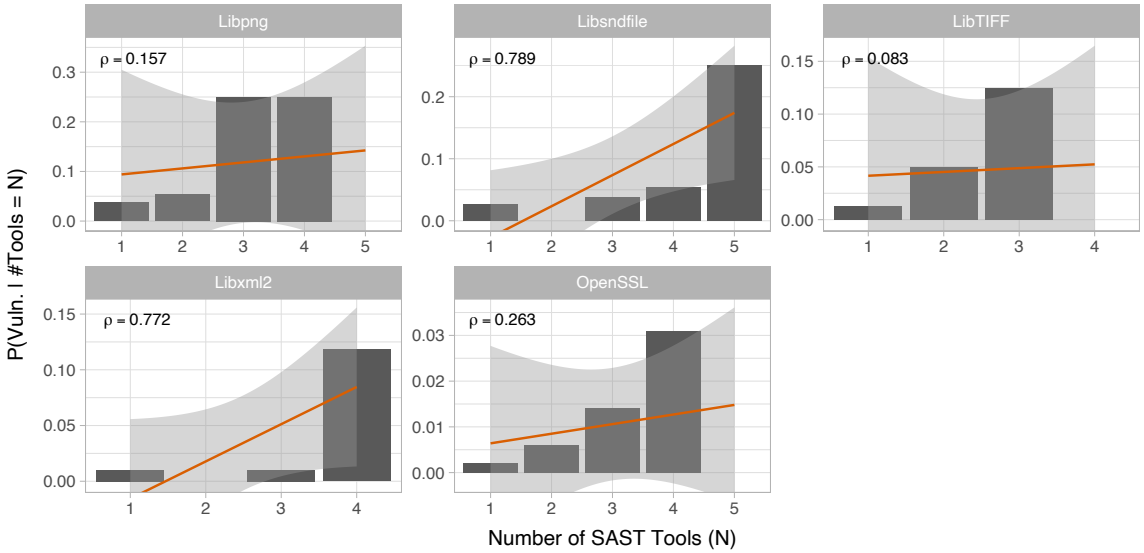


Figure 7.5.: Relationship between the number of SAST tools that flagged a function and the conditional probability of that function being vulnerable in each subject program (as appeared in [171]). The correlation strength between these two variables is measured by the Pearson correlation coefficient ρ.

In contrast, the same analysis for the SAST flag density heuristic reveals no positive correlation for all subjects, thus invalidating this heuristic, at least at the function level. For brevity, we omitted the corresponding plots from the thesis.

Summary (SAST-Based Vulnerability Heuristics)

Our preliminary study shows that functions flagged by multiple SAST tools are generally more likely to be vulnerable, regardless of the density of flagged lines. Therefore, we solely use the SAST tool consensus heuristic in Definition 7.2 to assign vulnerability scores to the TBBs.

Optimal Initial Interval Length

To find an optimal initial interval length and also to evaluate the effectiveness of our dynamic target scheduling against directed fuzzing with static targets, we compare SASTFuzz against WindRanger (static targets) using different initial interval lengths. Specifically, we compare (i) the number of covered TBBs (as identified by the employed SAST tools) and (ii) the numerical difference in detected bugs.

Figure 7.6 on the next page shows the average TBB coverage of the different SASTFuzz configurations, as well as that of WindRanger, throughout the fuzzing campaigns in each subject program. To evaluate the *effectiveness gains* of our fuzzer in terms of metric (i), we use the Vargha-Delaney $\hat{A}_{12}$ measure[5] [8] to compare the average TBB coverage at different

[5]For example, $\hat{A}_{12} = 0.5$ indicates no difference between the effectiveness of SASTFuzz and WindRanger, while deviations from 0.5 indicate different effect sizes.

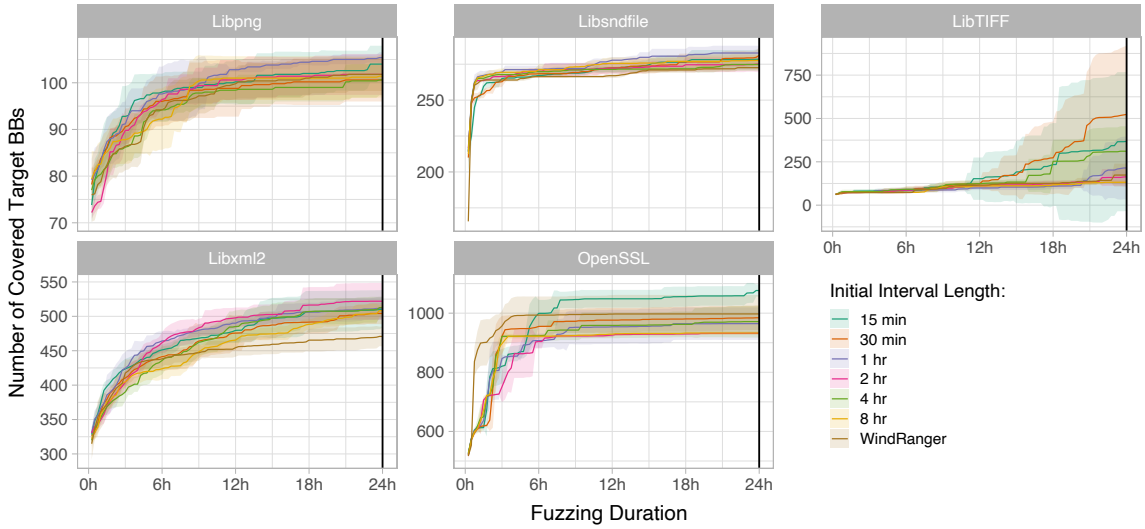


Figure 7.6.: Target basic block (TBB) coverage of SASTFuzz (dynamic targets) with different initial intervals and that of WindRanger (static targets) in each subject program (as appeared in [171]). The error bands show the standard deviation around the arithmetic means (solid lines).

points during the campaigns and across all subjects. For metric (ii), we compute the difference in the total number of unique bugs detected ΔB (i.e., the sum over all campaigns and subjects). The results of both analyses are shown in Table 7.2.

Table 7.2.: Effectiveness gains.[6]

Init. Interval	$\hat{A}_{12}$	ΔB
15 min	0.900	0
1 hr	0.692	+2
30 min	0.635	+2
2 hr	0.598	0
4 hr	0.590	0
8 hr	0.515	0

In terms of TBB coverage, our dynamic target scheduling—regardless of the initial interval chosen—consistently outperforms directed fuzzing with static targets, with shorter intervals yielding better results. While the campaigns with 15-minute initial intervals show the largest effectiveness gain ($\hat{A}_{12} = 0.9$), they also result in less thorough fuzzing due to early TBB deactivation, as fewer input hits are required to complete them (see Algorithms 7.1 and 7.2 on page 96). Here, only one bug (in Libsndfile) is detected, which is also found by WindRanger ($\Delta B = 0$). Similarly, campaigns with 2-hour, 4-hour, and 8-hour initial intervals do not find any additional bugs, even though they cover slightly more TBBs than WindRanger, because they do not reach the actually vulnerable TBBs within the allocated fuzzing time. In contrast, starting SASTFuzz with 30-minute or 1-hour intervals leads to significantly higher TBB coverage and more thorough fuzzing, resulting in the discovery of three unique bugs (in Libsndfile), two more than WindRanger ($\Delta B = +2$).

[6] This table previously appeared in [171].

Summary (Initial Interval Length)

Our preliminary study shows that an initial interval length of one hour provides the optimal balance between TBB coverage and bug detection. Using this configuration, SASTFuzz not only covers significantly more TBBs than WindRanger but also uncovers more unique security bugs.

7.4.3. Discussion

As shown at the beginning of this chapter, the SAST tools introduce only a relatively small runtime overhead of about 15 minutes, which can probably be further reduced by running them incrementally on code changes. This is especially useful for CI setups with strict resource constraints.

The selected analyzers effectively identify (potentially) vulnerable functions while, on average, flagging less than a third of all functions as problematic. This ratio decreases even further at the BB level, the granularity of our fuzzing targets, significantly reducing the code scope for fuzzing. Also, our proposed SAST tool consensus heuristic allows us to approximate the vulnerability probability of TBBs and, thus, to improve the target prioritization and resource distribution during fuzzing.

Regarding directed fuzzing, our dynamic target scheduling consistently outperforms the static-target approach used by WindRanger (and AFLGo) in reaching TBBs. Thereby, we find that an initial interval of one hour provides an optimal balance between TBB coverage and bug detection. However, for CI environments, shorter intervals (e.g., 5 to 15 minutes) may be better suited to maximize target coverage for the given time constraints.

7.5. Evaluation

In this section, we evaluate the bug-finding effectiveness, efficiency, and diversity of SASTFuzz compared to other widely used fuzzers. We perform a detailed analysis of the results, thereby examining the security bugs our fuzzer exclusively found, the bugs it missed, and the possible reasons for this. We also discuss the implications of these results and highlight the capabilities of SASTFuzz as well as its dependence on the accuracy of SAST tools.

7.5.1. Setup

Research Questions

This evaluation aims to show the superior effectiveness of our fuzzing approach compared to coverage-based fuzzing and existing vulnerability-guided approaches. Therefore, we answer the following *research questions* through empirical evaluations:

RQ 4a: **SASTFuzz Effectiveness.** How many new security bugs does SASTFuzz find compared to the baseline fuzzers?

RQ 4b: SASTFuzz Bug-Finding Diversity. What is the overlap and difference between the bugs found by SASTFuzz and those found by the baseline fuzzers?

RQ 4c: SASTFuzz Efficiency. How fast does SASTFuzz find bugs that were also found by the baseline fuzzers?

RQ 4d: Post-Bug-Detection Analysis. To what extent does our vulnerability-guided fuzzing approach contribute to the bugs found only by SASTFuzz, as well as to those missed by our fuzzer?

The first two research questions evaluate the bug-finding effectiveness of SASTFuzz from two perspectives: (i) the total number of new security bugs found and (ii) the diversity of bugs found relative to baseline fuzzers. This provides insights into the accuracy of our vulnerability-oriented code coverage metric in reflecting fuzzing effectiveness compared to the metrics integrated into the baseline fuzzers, especially regular code coverage. The third question examines the bug-finding efficiency of SASTFuzz and, thus, the accuracy of our metric in reflecting fuzzing efficiency. Although we performed a similar analysis in the previous chapter, we now approach it from a constructive rather than an analytical perspective while also including other vulnerability-oriented metrics. Finally, the fourth research question explores the limitations of SASTFuzz, specifically, whether and how quickly the bugs missed exclusively by our fuzzer could be detected if we had a more accurate prediction of the vulnerable code. This analysis aims not only to understand the impact of false-positive and -negative targets on the fuzzing performance but also to assess the future potential of SASTFuzz as vulnerability prediction techniques improve over time.

Subject Programs

To realistically and accurately evaluate our fuzzer against the baseline fuzzers, we follow Klees et al.'s guideline [144]: "The ultimate measure of a fuzzer is the number of distinct bugs that it finds. If fuzzer *A* generally finds more bugs than baseline *B*, then we can view it as more effective." Thereby, we even go one step further and—instead of comparing the fuzzers on programs with known bugs—assess their ability to detect new, previously unknown security bugs on the latest versions (as of August 2023) of 19 open-source programs. This way, we avoid in our evaluation *overfitting* (i.e., adapting tools to the known bugs) and *survivorship bias* (i.e., under- or overrating tools that found the original bugs) [26].

We randomly selected the subject programs from different application domains, with the only constraint being that we could get the fuzzers to run on them. Specifically, we include the following subjects: Discount, Jansson, jq, MyHTML, Parson, PDFio, and MDB Tools (data parsing); FLAC, Gifsicle, LibGD, libwebp, OpenJPEG, and Opus (multimedia processing); libmodbus and libsodium (networking and security); MIR, Pocketlang, and Zydis (programming languages); raylib (game programming).

Table 7.3 on the facing page shows the programs where at least one of the employed fuzzers could find a bug, thus forming the final set of subject programs for our evaluation. Each bug discovery is a strong indicator of a new security vulnerability and has, therefore,

Table 7.3.: Subject programs employed for evaluating the performance of our fuzzing approach (as appeared in [171]).

Subject	Commit	# Lines	# Basic Blocks	# Crashes	# Bugs
Discount	`a096c1a`	3,329	3,210	2,398	1
MIR	`928e28f`	17,496	13,668	16,228	50
MyHTML	`4f98bb1`	6,933	4,203	1,177	2
Parson	`b800e9d`	1,739	1,369	1,147	6
Pocketlang	`e316d53`	8,110	6,014	13,722	30
raylib	`557aeff`	19,840	10,655	10	1
Total		57,447	39,119	34,682	90

been reported to the respective developers. Several of these bugs have thereby already been confirmed as vulnerabilities.

Fuzzers

We choose AFL [326] (version 2.56b) as one of our baseline coverage-based fuzzers, which has been widely used by organizations such as Google to uncover thousands of vulnerabilities in critical OSS [100]. Since SASTFuzz is based on AFL via AFLGo and WindRanger, this allows us to compare *methodologies* rather than technical implementations [251], and thus how our vulnerability-guided fuzzing approach performs against coverage-based fuzzing. We also use AFL++ [86] (4.30c) as a baseline fuzzer, which is a widespread, community-driven fork of AFL that integrates the latest fuzzing research, making it the current state of coverage-based greybox fuzzing.

Furthermore, we compare SASTFuzz to three advanced vulnerability-guided fuzzers, TortoiseFuzz [305] (v0.1; based on AFL), ParmeSan [222] (no version specified), and FishFuzz [334] (no version specified; based on AFL++). They have been published at top-tier research conferences, with their source code openly available on GitHub, unlike most other directed fuzzers. TortoiseFuzz targets memory-accessing code, while ParmeSan and FishFuzz focus on sanitizer-instrumented code. However, unlike TortoiseFuzz and ParmeSan, FishFuzz—related to our target scheduling—dynamically prioritizes the 20% of fuzzing targets with the lowest execution frequency. All three fuzzers have demonstrated superior bug finding over many widely used fuzzers [222, 305, 334], thus making them ideal baseline candidates for our evaluation.

Seeds, Duration and Repetitions

As for the seed inputs, we use non-empty corpora sourced from different GitHub repositories[7] that contain files corresponding to the various formats of the subject programs.

[7] Seed corpora: `https://github.com/google/oss-fuzz`, `https://github.com/dvyukov/go-fuzz-corpus`, `https://github.com/strongcourage/fuzzing-corpus`

Table 7.4.: Bug-finding effectiveness of SASTFuzz and each baseline fuzzer in every subject program (as appeared in [171]), measured as the arithmetic mean μ with standard deviation and the total number of found security bugs Σ.

	SASTFuzz		AFL		AFL++		TortoiseFuzz		ParmeSan		FishFuzz	
Subject	μ	Σ	μ	Σ	μ	Σ	μ	Σ	μ	Σ	μ	Σ
Discount	1.0 ± 0.0	1	0.8 ± 0.4	1	1.0 ± 0.0	1	1.0 ± 0.0	1	0.0 ± 0.0	0	1.0 ± 0.0	1
MIR	21.0 ± 2.9	37	19.6 ± 2.6	32	12.9 ± 4.1	29	19.2 ± 3.0	33	-	-	23.0 ± 3.7	38
MyHTML	1.1 ± 0.3	2	1.0 ± 0.0	1	1.0 ± 0.0	1	1.2 ± 0.4	2	0.0 ± 0.0	0	1.0 ± 0.0	1
Parson	5.8 ± 0.4	6	6.0 ± 0.0	6	5.9 ± 0.3	6	5.9 ± 0.3	6	0.0 ± 0.0	0	5.6 ± 0.7	6
Pocketlang	12.1 ± 2.0	23	11.0 ± 1.4	19	10.6 ± 1.6	19	10.4 ± 1.1	16	3.6 ± 1.0	7	10.2 ± 1.0	19
raylib	0.3 ± 0.5	1	0.7 ± 0.5	1	0.0 ± 0.0	0	0.0 ± 0.0	0	0.0 ± 0.0	0	0.0 ± 0.0	0
Total		70		60		56		58		7		65

Moreover, we run each fuzzing campaign for 48 hours and repeat it 10 times, allowing us to compute average metric values to account for the randomness in fuzzing [144].

Bug Deduplication

Unlike the previous fuzzing experiments performed on the HPC, where AddressSanitizer was prohibited due to (potentially) excessive memory consumption, this time, we use a standard workstation for the experiments where we could enable sanitization for better bug detection. In total, the fuzzing campaigns triggered 34,682 program crashes (see Table 7.3 on the previous page). Since manually deduplicating these many crashes would be impractical, we use the approach described in Section 2.2.3 on page 23 to automatically deduplicate crashes into unique bugs. However, instead of just comparing the top-three stack frames of the crashing program executions, we now compare the top-five stack frames as well as the respective vulnerability types reported by the sanitizer, allowing for even more precise deduplication. Ultimately, this deduplication yields 90 unique, previously unknown security bugs, which we then manually reviewed again in order to confirm the validity of our approximation.

Infrastructure

We use the same hardware and software infrastructure as for the preliminary study (see Section “Infrastructure” on page 100).

7.5.2. Results

Note that despite extensive troubleshooting efforts, ParmeSan aborts the fuzzing campaign prematurely when executed on MIR, which is why we have no evaluation data for ParmeSan in this subject.

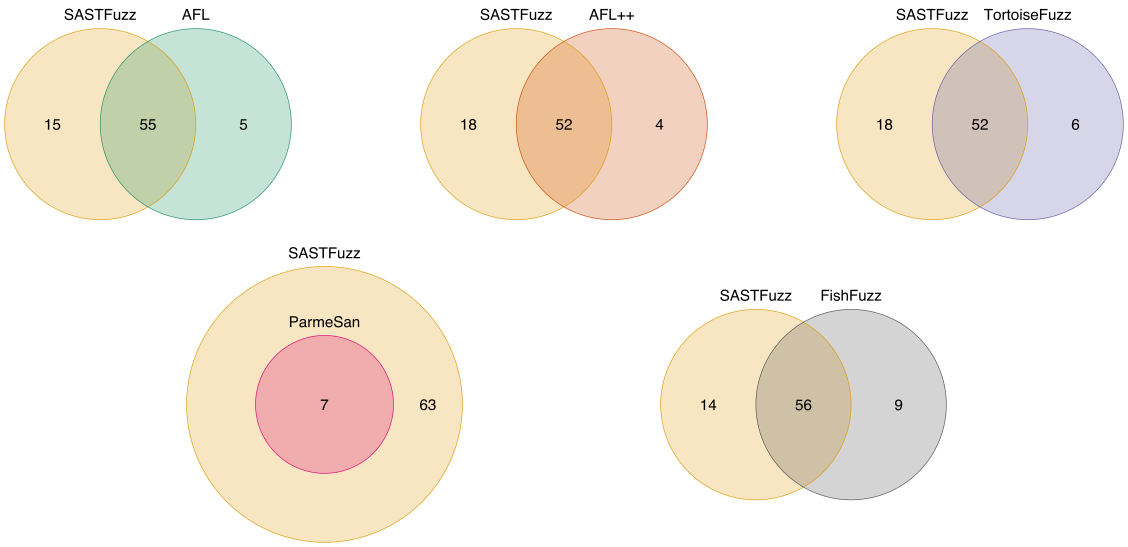


Figure 7.7.: Bug-finding diversity between SASTFuzz and each baseline fuzzer (as appeared in [171]), measured as the total number of identical and different security bugs found across all subject programs.

SASTFuzz Effectiveness

Table 7.4 on the facing page shows, for each subject program, the arithmetic mean, along with the standard deviation, and the total number of new security bugs found by SASTFuzz and the baseline fuzzers AFL, AFL++, TortoiseFuzz, ParmeSan, and FishFuzz. Regarding the arithmetic means, SASTFuzz achieves equal or higher average detection rates in Discount and Pocketlang. However, in MIR, MyHTML, Parson, and raylib, our fuzzer has slightly lower average rates than the best-performing baseline fuzzer, although, except in MIR, it detects at least the same or even higher total number of bugs. This indicates a slightly greater variation in SASTFuzz's effectiveness across the experiment repetitions. Regarding the total bug counts, SASTFuzz outperforms the baseline fuzzers in five out of six subjects, always matching or exceeding the effectiveness of the next best fuzzer. For example, in Pocketlang, SASTFuzz finds four more bugs than AFL, four more than AFL++, seven more than TortoiseFuzz, 16 more than ParmeSan, and four more than FishFuzz. The only exception is MIR, where FishFuzz detects one more bug than our fuzzer.

Summary (RQ 4a)

Our evaluation shows that SASTFuzz is more effective than the baseline fuzzers at detecting security bugs, uncovering 10 more previously unknown bugs in real-world programs than AFL, 14 more than AFL++, 12 more than TortoiseFuzz, 63 more than ParmeSan, and five more than FishFuzz. However, SASTFuzz's fuzzing campaigns have a slightly larger variation in bug-finding effectiveness across the campaign repetitions than those of the baseline fuzzers.

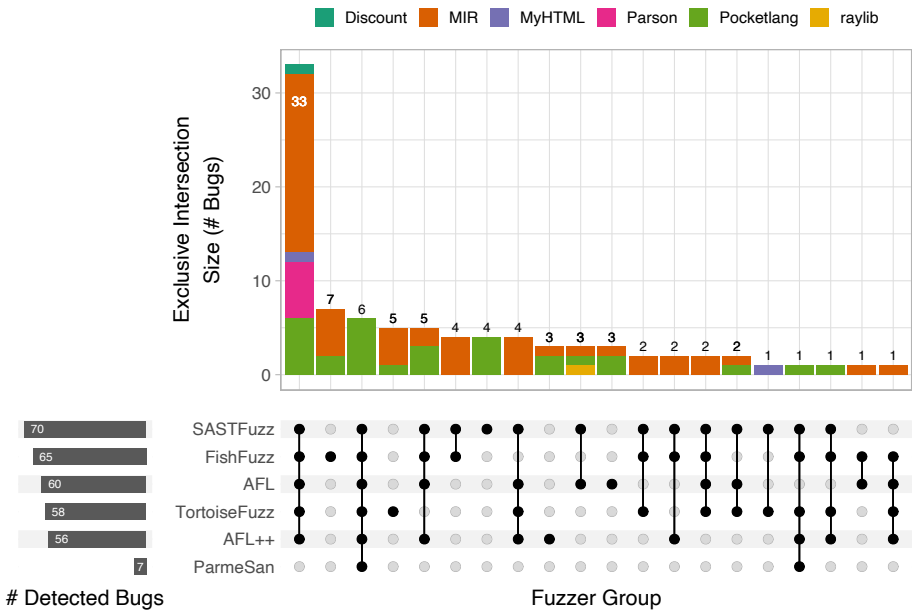


Figure 7.8.: Bug-finding diversity between different fuzzer groups, measured as the number of overlapping security bugs detected exclusively by the respective group (including the individual fuzzers) across all subject programs.

SASTFuzz Bug-Finding Diversity

Figure 7.7 on the previous page shows the overlaps and differences in the security bugs found between SASTFuzz and the baseline fuzzers across all subject programs in the form of Venn diagrams. For instance, compared to the best coverage-based fuzzer, SASTFuzz detects 15 bugs not found by AFL—10 more than the five found only by AFL. Compared to the best defect-guided baseline fuzzer, our fuzzer detects 14 bugs not found by FishFuzz—five more than the nine found exclusively by FishFuzz. Notably, SASTFuzz and FishFuzz account together for 79 out of all 90 bugs (87.8%) discovered during our evaluation.

Moreover, Figure 7.8 shows the number of overlapping security bugs detected exclusively by different fuzzer groups across all subject programs in the form of an UpSet diagram [158]. It shows that SASTFuzz finds four distinct bugs that are missed by all other fuzzers—one more than the three bugs exclusively found by AFL and AFL++, respectively. Obviously, this is also four more bugs compared to ParmeSan, which does not uncover any bugs that the others miss. However, it is still less than the five and seven bugs exclusively detected by TortoiseFuzz and FishFuzz, respectively.

Summary (RQ 4b)

Our evaluation shows that SASTFuzz detects a wider variety of security bugs than the baseline fuzzers, uncovering three times more different bugs than AFL (15 vs. 5), 4.5 times more than AFL++ (18 vs. 4), three times more than TortoiseFuzz (18 vs. 6), and about 1.6 times more than FishFuzz (14 vs. 9). Moreover, it finds all bugs detected by ParmeSan plus 63 additional ones. It also finds four bugs that all others miss—one more than AFL and AFL++, respectively, four more than ParmeSan, but also one and three bugs less than TortoiseFuzz and FishFuzz, respectively.

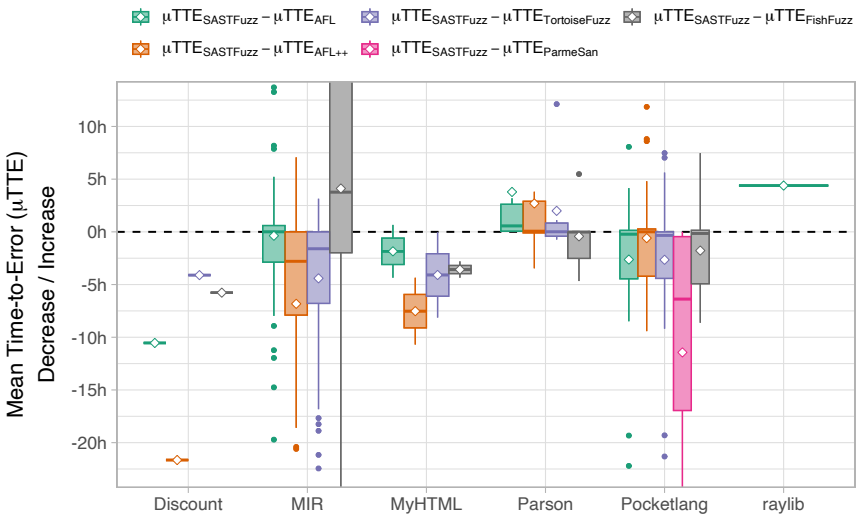


Figure 7.9.: Bug-finding efficiency of SASTFuzz relative to each baseline fuzzer (as appeared in [171]), measured as the average difference in time-to-error (TTE) per security bug in every subject program. Fuzzers that did not find any bugs in a subject are excluded from the comparison. Individual fuzzing campaigns that missed a bug are assigned a TTE of 48 hours. The diamonds (⋄) indicate the arithmetic means.

SASTFuzz Efficiency

Figure 7.9 shows the differences in mean time-to-error (TTE), abbreviated as μTTE, per detected bug between SASTFuzz and the baseline fuzzers in each subject program. Negative values thereby indicate a faster bug detection by SASTFuzz, while positive values indicate a faster detection by the respective baseline fuzzer. For fuzzing campaigns that miss a certain bug, we assign a TTE of 48 hours (i.e., the full campaign length).

Compared to AFL, SASTFuzz detects the bugs in Discount, MIR, MyHTML, and Pocketlang, on average, 10.5, 0.4, 1.8, and 2.6 hours faster, respectively. However, in Parson and raylib, AFL is more efficient, finding the bugs 3.8 and 4.4 hours earlier. The time savings of SASTFuzz are even more pronounced when compared to AFL++. On average, SASTFuzz is 21.6, 6.8, 7.5, and 0.6 hours faster in Discount, MIR, MyHTML, and Pocketlang, respectively. Notably, AFL++ does not detect any bugs in raylib, making a direct comparison impossible. The only case where AFL++ outperforms SASTFuzz is in Parson, where it finds bugs 2.7 hours earlier.

Moreover, SASTFuzz proves to be more efficient than the vulnerability-guided baseline fuzzers. It outperforms TortoiseFuzz by an average of 2.7 to 4.4 hours in four out of the five subject programs where TortoiseFuzz detects at least one bug, although it is two hours slower in Parson. SASTFuzz is also significantly faster than ParmeSan, finding the bugs in Pocketlang 11.4 hours earlier. In the other subjects, the average time savings are above 20 hours due to the many bugs missed by ParmeSan (not shown in the boxplot). Compared to FishFuzz, SASTFuzz detects the bugs in four out of the five subjects where FishFuzz finds any bugs, on average, 0.4 to 5.8 hours faster. One exception is MIR, where FishFuzz discovers the bugs 4.1 hours earlier than SASTFuzz.

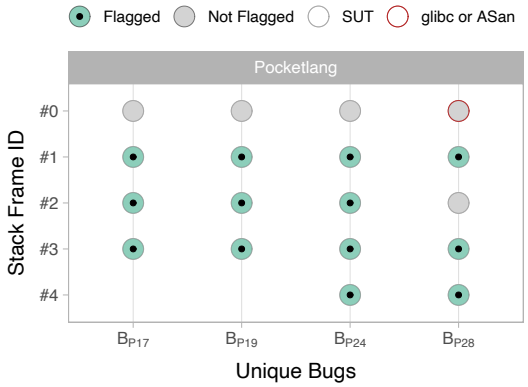


Figure 7.10.: Top-five (SAST-flagged) stack frames on the crashing execution trace of the security bugs found only by SASTFuzz (as appeared in [171]). Frame #0 represents the function in which the bug manifests and the execution crashes, which may occur outside the system under test (SUT) (outlined in red).

Summary (RQ 4c)

Our evaluation shows that SASTFuzz is more efficient than the baseline fuzzers at detecting security bugs. During 48-hour fuzzing campaigns, it uncovers the bugs in most subject programs, on average, 0.4 to 10.5 hours faster than AFL and 0.6 to 21.6 hours faster than AFL++, although being up to 4.4 hours slower in some cases. SASTFuzz is also, on average, 2.7 to 4.4 hours faster than TortoiseFuzz, 11.4 hours faster than ParmeSan, and 0.4 to 5.8 hours faster than FishFuzz. Only in two subjects is either TortoiseFuzz or FishFuzz faster by 2.0 and 4.1 hours, respectively.

Post-Bug-Detection Analysis

In this section, we examine (i) whether the bugs found exclusively by SASTFuzz result from the fuzzing targets identified by the SAST tools and (ii) why our fuzzer misses certain bugs found by the baseline fuzzers.

Bugs Found Only by SASTFuzz. Figure 7.10 shows the top-five stack frames (i.e., the functions in the execution trace leading to the vulnerable code) of the bugs found exclusively by SASTFuzz, thereby highlighting the frames flagged by the SAST tools. As shown in the figure, our fuzzer found four bugs in Pocketlang that all the other fuzzers missed. While these bugs share some execution frames, they ultimately manifest in different functions within frame #0, which, however, was not flagged by the analyzers. In contrast, the other frames were consistently flagged by the SAST tools, effectively guiding SASTFuzz to the vulnerable code. In particular, the segmentation violation (segv) bugs B_{P17} and B_{P19} exhibit a pattern where three out of four frames (maximum trace length) were flagged. A similarly high ratio appears for the segv B_{P24}, where four out of five frames were flagged. For the stack overflow B_{P28}, although frame #0 is outside the analysis scope of the SAST tools, three out of four frames in the execution trace were flagged.

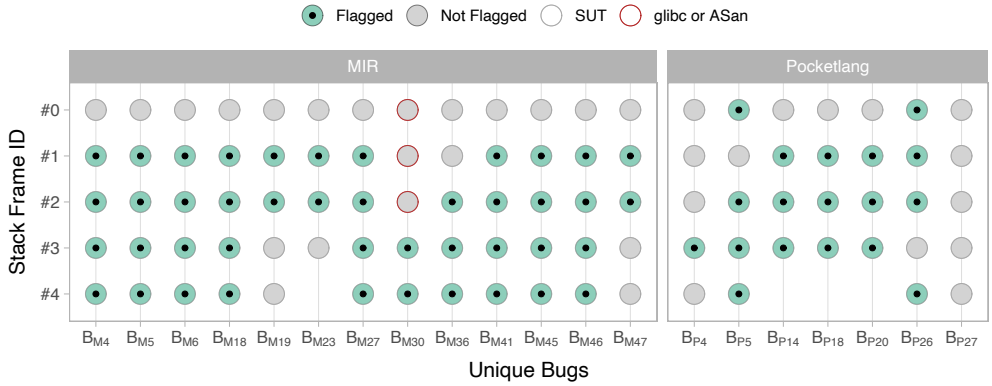


Figure 7.11.: Top-five (SAST-flagged) stack frames on the crashing execution trace of the security bugs not found by SASTFuzz. Frame #0 represents the function in which the bug manifests and the execution crashes, which may occur outside the system under test (SUT) (outlined in red).

Bugs Found Only by the Baseline Fuzzers. Figure 7.11 shows a similar graph, but this time for the bugs missed by SASTFuzz. Here, only in the case of the stack overflow B_{P27} in Pocketlang, all of the top-five stack frames were not flagged by the SAST tools. This is because a single unflagged function is called recursively several times before the bug is triggered, causing SASTFuzz to ignore this false-negative target during fuzzing. For the segmentation bugs B_{M19}, B_{M23}, and B_{M47} in MIR, as well as B_{P4} in Pocketlang, the analyzers flagged significantly fewer frames, reducing the thoroughness with which SASTFuzz targets these bugs. Interestingly, the analyzers flagged most of the relevant frames for the remaining undetected bugs. However, replaying the fuzz inputs stored in SASTFuzz 's queue during the experiments reveals that—despite following the right direction—they never actually reach the functions where these bugs manifest and eventually cause the crash. This may be explained by the numerous (false-positive) SAST tools findings used as fuzzing targets, forcing SASTFuzz to limit the number of executions per target in order to maximize overall target coverage. This then results in a premature direction shift in the test execution, preventing our fuzzer from reaching or triggering the bugs.

These findings are further supported by Table 7.5 on the following page, which shows the bug-finding performance[8] of SASTFuzz when filtering out the false-positive target locations and directing SASTFuzz to the top-five stack-frame locations on the crashing execution trace of the previously missed security bugs—essentially simulating a perfect vulnerability prediction. Now, using this set of fuzzing targets, SASTFuzz detects 13 of the 20 previously missed bugs (65%). Notably, it detects the bugs B_{M5}, B_{M6}, B_{M19}, and B_{M47} in MIR 7 minutes, 1.8 hours, 31.2 hours, and 30.9 hours faster, respectively, than the fastest baseline fuzzer during 48-hour fuzzing campaigns. Hence, these results show not only the impact of false positives on our fuzzing approach but also highlight the future bug-finding potential of SASTFuzz—its performance will *automatically improve* as the accuracy of SAST techniques improves.

[8]For this analysis, we used the same fuzzing configuration as well as software and hardware infrastructure as in the previous experiments.

Table 7.5.: Bug-finding performance of SASTFuzz when using the top-five stack-frame locations on the crashing execution trace of the previously missed security bugs as fuzzing targets (as appeared in [171]). The last column indicates the increase or decrease in time-to-error (TTE) compared to the baseline fuzzers that found the respective bugs the fastest.

Subject	Bug ID	Bug Type	Detected	ΔTTE [1]
MIR	B_{M4}	Segv	✓	2 hr 38 min
	B_{M5}	Segv	✓	-0 hr 07 min
	B_{M6}	Segv	✓	-1 hr 45 min
	B_{M18}	Segv		-
	B_{M19} [2]	Segv	✓	-31 hr 12 min
	B_{M23}	Segv	✓	7 hr 53 min
	B_{M27}	Segv	✓	3 hr 08 min
	B_{M30} [2]	Segv	✓	5 hr 16 min
	B_{M36}	Segv	✓	4 hr 33 min
	B_{M41}	Segv		-
	B_{M45}	Segv	✓	2 hr 24 min
	B_{M46}	Segv	✓	9 hr 29 min
	B_{M47} [2]	Segv	✓	-30 hr 55 min
Pocketlang	B_{P4}	Segv		-
	B_{P5}	Segv		-
	B_{P14}	Segv	✓	2 hr 23 min
	B_{P18}	Segv		-
	B_{P20}	Segv		-
	B_{P26}	Segv	✓	34 hr 24 min
	B_{P27}	Stack-overflow		-

[1] $\Delta TTE = TTE_{SASTFuzz} - TTE_{Baseline}$
[2] The crashes generated by SASTFuzz were intercepted at stack frame #0 by AddressSanitizer rather than the operating system, as was the case with the crashes of the respective baseline fuzzer. However, all of the other frames match, including the vulnerability type.

Summary (RQ 4d)

Our evaluation shows that the employed SAST tools flagged most of the top-five stack frames on the crashing execution trace of the security bugs exclusively detected by SASTFuzz, effectively guiding our fuzzer to the vulnerable code. In contrast, the bugs missed by SASTFuzz were mainly due to the large number of (false-positive) targets, forcing SASTFuzz into less exhaustive fuzzing. However, our post-bug-detection analysis shows that these bugs are likely detectable with our vulnerability-guided fuzzing approach as SAST techniques become more accurate.

7.5.3. Discussion

SASTFuzz, which implements our vulnerability-guided fuzzing approach, detects previously unknown security bugs[9] *more effectively* than the state-of-the-art baseline fuzzers, which implement coverage-based or sanitizer-/memory-access-guided approaches. It also uncovers more different bugs than the baseline fuzzers, making SASTFuzz a valuable addition to existing fuzzer ensembles. Ultimately, these results demonstrate the improved accuracy of our metric in reflecting fuzzing effectiveness.

Our post-bug-detection analysis confirms that the superior bug finding is mainly due to the employed SAST tools, which effectively guide our fuzzer to the potentially vulnerable code. Thereby, as SAST techniques continue to improve, SASTFuzz will *automatically* become more powerful.

Furthermore, while SASTFuzz already detects the bugs in most subject programs *more efficiently* than the baseline fuzzers, optimizing the heavy-weight distance computations could further speed up the bug finding.

7.6. Threats to Validity

To minimize threats to *external validity*, we conducted a large-scale empirical evaluation comparing SASTFuzz with two coverage-based fuzzers and three vulnerability-guided fuzzers. Moreover, we employed 19 different real-world subject programs with unknown security bugs, thus preventing our evaluation from potential tool overfitting and bug survivorship bias.

In terms of *internal validity*, the numerous program crashes forced us to automate the process of bug deduplication. As before, we used a well-established heuristic from the fuzzing domain to deduplicate crashes into unique bugs based on the similarity of the stack frames in the crashing execution traces. This time, however, we compared the top-five stack frames along with the vulnerability type reported by AddressSanitizer rather than just the top-three frames. This resulted in a precise deduplication for the respective subject programs, as confirmed manually.

Furthermore, we did not perform statistical tests in our evaluation since achieving statistical significance in bug-based benchmarking is difficult due to the scarcity of real-world security bugs (i.e., limited data points). However, the considerably higher number of distinct bugs found by SASTFuzz in extensively tested software, compared to each baseline fuzzer, inspires confidence in our results and conclusions.

Finally, given the substantial time and hardware costs associated with fuzzing, we—similar to Hazimeh et al. [114]—compromised in our evaluation on the number of fuzzing campaign repetitions (i.e., 10 instead of 20 or 30) in favor of a longer campaign length (i.e., 48 instead of 24 hours). We believe this is a reasonable tradeoff because new and more complex security bugs can often only be found with fuzzing efforts that last longer than the usual 24 hours.

[9] Note that a single new vulnerability can have far-reaching consequences, as illustrated by the high-severity vulnerability in libwebp (CVE-2023-4863), which affected millions of devices [113].

7.7. Summary

In this chapter, we introduced our fuzzing approach that integrates our vulnerability-oriented code coverage metric as a test-optimization criterion into a directed fuzzer to target potentially vulnerable code identified by SAST tools. Our approach also includes advanced mechanisms for dynamically scheduling fuzzing targets and switching between coverage-based and vulnerability-directed fuzzing in order to mitigate the impact of SAST tool limitations on fuzzing performance. In a preliminary study, we showed that our dynamic target scheduling outperforms directed fuzzing with static targets in terms of target coverage and bug finding. Moreover, in a large-scale evaluation, we showed that SASTFuzz not only uncovers more and different previously unknown security bugs but also detects bugs much faster than the coverage-based and vulnerability-guided baseline fuzzers. Ultimately, this demonstrates the superior accuracy of our fuzzing metric in reflecting fuzzing effectiveness compared to the respective baseline metrics.

Part IV.

Related Work and Conclusion

8. Related Work

This chapter presents related work on measuring fuzzing effectiveness, including existing (but inadequate) solutions for each sub-problem that had to be addressed for our solution approach. Parts of this chapter have appeared in a submission [171] and peer-reviewed publications [168–170], first-authored by the author of this thesis.

8.1. Static and Predictive Bug Detection

In this section, we review related work on statically identifying potentially vulnerable source code, which is important for our metric to narrow down the relevant code considered during fuzzing. Thereby, we highlight the research gaps identified and addressed in this dissertation, which aim to evaluate the effectiveness of SAST tools and develop an alternative, more reliable vulnerability prediction approach.

8.1.1. SAST Tool Evaluation

SAST tools are commonly used to quickly identify potentially vulnerable code [17]. While the issue of false positives has been extensively researched [5, 56, 103, 127, 135, 137, 139, 204, 254, 294], the detection rate (and thus false negatives) of such tools has received less attention. The importance of such an analysis is thereby highlighted by the community project CVE Benchmark [3], initiated by the Open Source Security Foundation[1], which provides a platform for evaluating (JavaScript) SAST tools against real-world vulnerabilities (i.e., CVEs). While there are also several empirical evaluations of Java bug scanners [105, 160, 289] (whose results may not generalize to C/C++ analyzers), only very few exist in the context of C/C++. The existing studies, however, mainly employ benchmark datasets, such as the Juliet dataset [27], consisting of small toy programs—often less than 100 LoC—that contain artificially introduced, easy-to-find security bugs [10, 46, 94, 102, 126, 138, 219, 295]. Accordingly, the complexity of these artificial bugs does not reflect that of real-world bugs [36, 93], thus providing limited insights into the true effectiveness of static analyzers. Below, we highlight our contribution of evaluating the effectiveness of modern C/C++ SAST tools using a representative set of security bugs by discussing the differences between our and existing evaluations.

Zitser et al. [340] evaluate five freely available SAST tools, ARCHER [314], BOON [296], PolySpace C Verifier [181], Splint [80], and UNO [119], against 14 known buffer-overflow vulnerabilities in three open-source projects. While this study depicts the state-of-the-art in 2004, it provides limited insights into the effectiveness of current SAST tools because these analyzers are now largely obsolete and no longer maintained. Hence, we evaluate

[1] https://openssf.org/

newer tools with more advanced SAST techniques. To gain reliable insights into their effectiveness, we evaluate them on 198 security bugs spread across 27 projects. However, this large experimental setup forced us to automate the evaluation using the function level as the detection granularity while Zitser et al. manually examined the flagged lines. Interestingly, they report similar detection rates of about 30% across the SAST tools.

Zheng et al. [335] conduct a similar yet different evaluation in the sense that they examine not only the bug-finding effectiveness but also the associated economic costs of three commercial SAST tools, FlexeLint (now PC-lint Plus) [128], Klocwork [271], and Illuma [249], on three programs within Nortel's network service software. However, their experimental setup does not provide insights about current free analyzers, so we include—besides a commercial SAST tool—five free tools. We evaluate these static analyzers on several programs from various application domains to draw more general conclusions about their effectiveness. Our detection rates are slightly lower but still in the same range as Zheng et al.'s rates, even though our automated evaluation is at the function level, as opposed to their manual line-level analysis.

Furthermore, as part of their study on combining static analysis traces with dynamic symbolic execution, Busse et al. [38] evaluated the effectiveness of two SAST tools, Infer [188] and Clang Static Analyzer [237], by manually analyzing their false and true positives across many different open-source projects. While their results confirm the problem of excessive false positives, they also show—as does our study—that static analyzers fail to detect many real-world vulnerabilities. Unlike their work, however, our evaluation includes five additional state-of-the-art analyzers, including a commercial tool, and uses an automated evaluation framework. This allows for future assessments of newer SAST tools on the same benchmark programs to track their advancements.

Finally, (i) Pereira and Vieira [231] and (ii) Croft et al. [62] also use an automated approach to evaluate the effectiveness of syntactic SAST tools. While study (i) evaluates Cppcheck [179] and Flawfinder [310] on a dataset with security bugs in various Mozilla software projects [7], study (ii) evaluates RATS [272] in addition to these two analyzers on real-world projects with known bugs. However, both studies do not fully depict the capabilities of current SAST tools because study (i) is limited to tools that work on raw source code (due to Mozilla's custom build process) and study (ii) focuses only on pattern-matching scanners. This excludes more sophisticated semantic tools such as Infer [188] and CodeQL [95], which attach to the build process to identify more complex bugs. Also, both studies consider a bug detected when an analyzer flags (any line in) the vulnerable source file. However, this can lead to overly optimistic detection rates, as shown by the rates of 84% (Cppcheck) and 36% (Flawfinder) in study (i) and 51% (Cppcheck), 63% (Flawfinder), and 41% (RATS) in study (ii), which strongly differ from our results.

Summary

We address Gap 1 (assessing SAST tool effectiveness on real-world programs) by empirically evaluating (i) current state-of-the-art (free and commercial) C/C++ instead of JavaScript or Java analyzers on (ii) open-source projects with known security bugs (of various types) instead of toy programs with artificial bugs, thereby using (iii) function instead of file level as detection granularity.

8.1.2. Vulnerability Prediction

Besides SAST, researchers have proposed several heuristics for detecting source code that is likely to contain security bugs, which are mainly based on code complexity [73, 187, 268, 337], code churn [207, 268, 336], and text or bug-pattern mining [140, 273, 297, 316–319]. Complexity-based heuristics such as the *McCabe* [183] and *Halstead* measures [108], *coupling & cohesion* [291], and control-flow nesting [236] have thereby proven particularly effective at identifying potentially vulnerable code.

Over time, numerous studies have sought to enhance static bug finding by incorporating various vulnerability heuristics, hereafter referred to as *features*, into predictive models using ML. These models often rely on several complexity-based features [55, 73, 78, 90, 115, 185, 199, 200, 208, 214, 269, 275, 313], which, however, are limited to certain, less complex types of bugs that are detectable via the surrounding code structure. In contrast, our approach incorporates features derived from SAST tools, which, as shown in our study, can detect more complex bugs that require logical reasoning to find them.

These limitations have led to the development of new defect prediction models that additionally incorporate code churn information [92, 155, 198, 206, 246, 270, 327, 339]. While code churn can be a valuable addition, the resulting model depends on accessible development history data, which limits its applicability to programs for which such data is available. Therefore, we decided to omit this class of features for now but may consider it for future work.

Other related studies suggest using features derived from text or bug-pattern mining [134, 257, 332] and data-flow analysis [148, 267]. Both feature classes are thereby largely covered by our SAST-based features. The former is subsumed by the employed syntactic static analyzers Flawfinder [310], Cppcheck [179], and Semgrep [262], while the latter is subsumed by the semantic analyzers Infer [188], CodeQL [95], and Clang Static Analyzer [237]. Apart from the feature set, our approach also differs from the referenced papers in that some of them train and/or evaluate the model on artificial instead of real-world codebases. Furthermore, most of these works predict potentially vulnerable code at the class, file, or module level, with the goal of minimizing the number of false positives. In contrast, we predict bug-prone code at the much finer function level, thereby specifically optimizing for fewer false negatives.

To the best of our knowledge, only (i) Nagappan and Ball [205], (ii) Gegick et al. [92], and (iii) Pereira et al. [230] integrate SAST-based features into their defect prediction models. All three studies use the density of lines flagged by SAST tools as a prediction feature, with study (ii) additionally including LoC and code churn-based features and study (iii) various complexity-based features.

Specifically, study (i) uses the static analyzers PREfix [37] and PREfast (a faster version of PREfix) [152] in order to derive the SAST flag density feature values. In their evaluation, conducted on 199 software components of Windows Server 2003 (each consisting of several thousands of LoC), they measure a strong positive correlation between the SAST flag density and the bug-finding probability. Thereby, their model achieves an 83% accuracy in classifying the components as vulnerable or non-vulnerable.

Regarding study (ii), the authors use the analyzers FlexeLint (now PC-lint Plus) [128] as part of their defect prediction model, which they evaluate on a proprietary telecommu-

nications software consisting of 25 components. Their results show true and false positive rates at component-level granularity of 100% and 8%, respectively. Although they report fewer false positives when including the SAST flag density feature, they observe a weaker correlation with bug detection than Nagappan and Ball.

Lastly, study (iii) evaluates their model, which is based on FLAWFINDER [310] and CPPCHECK [179], on a dataset [7] built from Mozilla bug reports. Their results show recall and precision scores at file-level granularity of 90% and 23%, respectively. Moreover, they report that both feature classes are required to achieve the best possible prediction in their experimental setup.

Different from all three studies, we employ six SAST tools, including FLAWFINDER and CPPCHECK, as well as the error detector ADDRESSSANITIZER [263], which allows us to introduce a novel heuristic, namely the consensus among tools about problematic code. We show that this heuristic is a more accurate indicator of bug-prone code than the SAST flag density, for which we found no positive correlation. Ultimately, our vulnerability prediction model achieves similar true and false positive rates of 90% and 50%, respectively; however, at the finer code granularity of functions, as opposed to the file- or component-level prediction in the studies (i-iii).

More recent approaches use deep learning (DL) to automatically learn the relevant features for defect prediction from the raw training data [159, 243, 253, 303, 304, 322], thus (supposedly) eliminating the need for manually defined defect features. However, such DL models tend to learn features that are entirely unrelated to the actual cause of the security bugs, as shown by several replication studies [43, 216, 278]. As a result, they are often significantly less effective in real-world applications than claimed in the papers and thus also perform worse than most models with predefined feature sets.

Summary

We address Gap 2 (combining SAST with code complexity heuristics to improve bug finding) by developing a vulnerability prediction model that (i) utilizes a set of manually defined features instead of automatically extracted ones. It also (ii) leverages a novel and very effective SAST-based feature derived from the consensus of multiple analyzers flagging a code region as problematic. Finally, the model allows us to (iii) accurately and reliably identify potentially vulnerable code at the level of functions rather than classes, source files, or even components.

8.2. Fuzzing Assessment and Optimization

In this section, we outline our contributions in terms of assessing and optimizing the performance of fuzzing with respect to related work, thereby focusing on existing approaches for stopping fuzzing campaigns (including technical solutions to achieve this) and vulnerability-guided fuzzing. Both areas reveal open research problems that can be addressed with our vulnerability-oriented code coverage metric, allowing us to demonstrate its improved accuracy in reflecting fuzzing (cost-)efficiency and effectiveness over existing metrics.

Several fuzzing metrics have been proposed that treat the SUT as a black box. They measure the difference in observed test input and/or output data, assuming that greater diversity leads to more extensive testing.

Existing input diversity metrics quantify the amount of different fuzz inputs produced within the input domain, usually specified by a set of constraints [279, 292] or a formal grammar [18, 276]. Obviously, this limits the applicability of these metrics to programs for which such a (domain-specific) input specification exists. Moreover, their implicit assumption that input diversity converges with program-behavior diversity can lead to test gaps since different inputs may execute identical program paths. Nguyen and Grunske [212] mitigate this thread by additionally including code coverage information, but unlike our metric, considers all code as equally important.

Regarding output diversity, the most common metric is the number of new bugs discovered [114], often using program crashes as a proxy for (unique) bugs. However, this can easily lead to incorrect fuzzing assessments due to the unknown number of bugs contained in the SUT, with crash-based metrics skewing the analysis even more since multiple crashes may result from the same bug [144, 170, 177]. To overcome these limitations, our metric treats the SUT as a white box, measuring the coverage of potentially vulnerable code.

8.2.1. Code Coverage Tracking

Tracking accurate code coverage during a fuzzing campaign, regardless of the fuzzer used, is a major technical challenge due to the high input throughput of modern fuzzers. Existing solutions [164, 189, 190] try to sample a representative subset of all generated fuzz inputs, such as those added to the fuzzer queue for further mutation, as in mutation-based fuzzing. To measure code coverage, these inputs are then replayed on another instance of the SUT compiled with coverage support, e.g., using Gcov [1] or llvm-cov [239]. However, this approach has several limitations.

First, it inhibits measuring how often certain code regions have been executed, which is important for identifying less thoroughly tested program parts.

Second, it relies on fuzzers that store coverage-increasing inputs in a directory (e.g., the fuzzer queue) so that they can be replayed later. Obviously, this necessitates that the sampled subset accurately represents the effectiveness of all fuzz inputs and, thus, that of the entire campaign. However, due to path collisions [63] in fuzzers that hash the tracked code coverage information [86, 89, 169, 299], not all coverage-increasing inputs are added to the queue and are therefore not stored in the respective directory. Consequently, when measuring code coverage based on these inputs, code that was covered during the actual campaign may now be missed. This overshadowing of coverage progress can lead to distorted measurements and, in the worst case, if this information is used to decide when to stop a campaign, to the premature termination of campaigns that would otherwise have found vulnerabilities.

Third, relying on a second, coverage-instrumented instance of the SUT adds significant hardware and synchronization overhead, making real-time code coverage tracking impractical in larger fuzzing setups.

For these reasons, our coverage analyzer instruments the same SUT instance that will later be fuzzed, which allows monitoring of the coverage progress independently of the employed fuzzer and in parallel to the campaign, without much overhead.

FUZZ-INTROSPECTOR [2] is another related tool that is less of a coverage analyzer and more of a fuzzing introspection platform that provides detailed static program and dynamic test-execution information of the fuzzing efforts. It helps practitioners to identify fuzzing blockers in the SUT's source code [91], i.e., hard-to-pass conditions that prevent the fuzzer from increasing code coverage. Hence, once identified, they allow for developing better test harnesses and for finding better fuzzer configurations. However, FUZZ-INTROSPECTOR relies on external coverage analyzers (e.g., GCOV or LLVM-COV) to obtain the code coverage information achieved by the campaign and thus suffers from the same drawbacks described above. Combining our analyzer with FUZZ-INTROSPECTOR could, therefore, be an interesting direction for future work.

Summary

We address Gap 3 (real-time monitoring of fuzzing code coverage progress) by developing a coverage analyzer that (i) collects fine-grained code coverage information. This involves directly instrumenting the SUT to (ii) provide real-time coverage information to our analyzer, enabling dynamic campaign termination based on our stopping criteria. Also, we (iii) track the code coverage of all fuzz inputs, thus protecting our analyzer from the path collision problem inherent in most mutation-/coverage-based fuzzers, which can distort the code coverage measurement.

8.2.2. Fuzzing Stopping Criteria

In general, very little research has been done so far on criteria for stopping fuzzing campaigns. In terms of evaluating and comparing fuzzers, Klees et al. [144] suggests a fixed campaign length of at least 24 hours because some fuzzing techniques become more effective later in the campaign as more inputs are added to the queue. Accordingly, recent fuzzer evaluations [114, 164, 177, 258] have adopted fuzzing durations ranging from 24 hours to 7 days. Interestingly, FuzzBench, Google's fuzzer benchmarking platform [189], opts for 23-hour campaigns instead. This choice is based on their observation that fuzzer rankings, as determined by code coverage achieved, remain largely unchanged between 23-hour and 7-day experiments while vastly reducing the associated hardware costs. The work of Ounjai et al. [223] takes this idea a step further: at the cost of additional experiments, they decrease the campaign length even further without sacrificing much of the fuzzer ranking accuracy with respect to the large-scale FuzzBench results. Hence, while their approach is not intended to completely replace large-scale experiments, it still allows for large time savings by occasionally running shorter evaluations. Although these studies provide valuable insights into the optimal campaign length for comparative fuzzer evaluations, using a fixed fuzzing time budget can lead to premature termination (i.e., missing vulnerabilities) or unnecessarily long campaigns (i.e., wasting time or hardware resources) when searching for new security bugs in the wild.

In practice, campaigns are typically stopped when the number of generated program crashes or covered code regions saturates for an extended period of time [287], which can lead to an inappropriate termination point. In contrast, our criterion considers the saturation of the covered, potentially vulnerable code, allowing (ineffective) campaigns to be terminated typically several hours earlier than either of the above criteria, with little risk of missing bugs. Alternatively, Böhme et al. [23] suggest stopping campaigns based on the estimated residual risk of security bugs in the SUT. Specifically, they conceptualize fuzzing as a sampling process and use various statistical methods [20] to estimate the probability of undiscovered bugs. Once the risk falls below a certain threshold, they recommend stopping the campaign because it becomes impractical to continue fuzzing beyond that point. However, unlike our stopping criterion, their approach is tied to the particular fuzzer being used. This dependency arises because their criterion must account for the adaptive bias [20, 23] inherent in all fuzzing strategies, where the probability of generating bug-revealing inputs increases as the campaign progresses.

Summary

We address Gap 4 (creating a cost-efficient fuzzing stopping criterion) by developing a vulnerability-oriented stopping criterion that, by terminating fuzzing campaigns when the amount of covered, potentially vulnerable code keeps saturating for a certain period, (i) adapts to the campaign's effectiveness, (ii) is fuzzer-independent, and (iii) outperforms existing criteria in terms of cost-efficiency.

8.2.3. Vulnerability-Guided Fuzzing

Böhme et al. [24] introduced the concept of directed greybox fuzzing, which has since inspired numerous advances [302] in (multi-)target distance computation [15, 47, 74, 124, 154, 166, 167] and seed prioritization & scheduling strategies [47, 124, 125, 141, 154, 162, 166, 167, 242, 265, 324, 336]. Moreover, new mutation operators have emerged that, for instance, dynamically adjust mutation granularity [12, 47, 323] and/or utilize data-flow analysis [74, 118, 124, 125, 141, 162] in order to improve directed fuzzing.

To reach more fuzzing targets in less time, recent studies filter out fuzz inputs that are unlikely to reach any targets by predicting their target reachability [15, 341] or by inferring logical conditions (invariants) from program executions to constrain the input generation [125]. Other approaches instead attempt to statically identify target-reachable program paths for input prioritization [312] or selective path instrumentation & exploration [141, 174], as well as to prune target-unreachable paths by discarding executions once they enter these paths [123, 215, 277].

Unlike these approaches, we dynamically enable and disable targets during fuzzing, an approach used only by Zheng et al. [334] so far. They focus on the 20% least-executed targets, thereby excluding unreached targets from the exploitation phase altogether. Our target scheduling differs in two ways: (i) we completely disable apparent false positives for the rest of the campaign, allowing our fuzzer to focus on other, more likely vulnerable targets, and (ii) we systematically include unreached targets in the exploitation phase to increase their chances of getting covered. As for the other optimizations, they are mostly complementary to ours and may be used in the future to further improve our approach.

Directed fuzzing is often coupled with vulnerability prediction techniques for automated target acquisition. Initial research in this area aimed at detecting specific types of vulnerabilities, such as use-after-free [213, 298], buffer and integer overflows [107, 132, 186], and DoS issues [156, 161, 233, 307]. Instead, we employ multiple advanced SAST tools that support the most common C/C++ vulnerabilities [168], which allows for detecting different vulnerability types with only a single fuzzer, thus reducing the associated execution costs.

Newer vulnerability-guided fuzzing approaches also support a wider range of vulnerabilities, both at the binary and source level. Binary-level approaches primarily utilize static code analysis [133, 213, 229] or ML & DL [163, 333] on the reverse-engineered SUT to identify interesting fuzzing targets, contrasting with our approach that operates on the source code. Existing source-level approaches use code churn metrics [329, 336], software patches [323], or bug tracker information [305, 323] in order to guide fuzzing towards potentially vulnerable code. Unlike these vulnerability heuristics, our static analyzers do not rely on any pre-existing artifacts other than the SUT's source code.

For instant target extraction, Du et al. [73] combine several code complexity metrics into a (closed-source) vulnerability prediction model. The authors also demonstrate the effectiveness of their complexity-guided fuzzing approach by detecting several vulnerabilities in the PHP codebase. Nonetheless, code complexity is generally limited to simple, syntactically detectable security bugs, often resulting in fewer bugs found compared to using more advanced techniques [305]. As a result, recent studies often leverage sanitizer checks [50] or sanitizer-instrumented instructions [222, 265, 334] to identify bug-prone code for fuzzing, as assertions have been shown to strongly correlate with test effectiveness [331]. The referenced papers present promising results, indicating that sanitizer-guided fuzzing can outperform coverage-based and other vulnerability-guided approaches. However, sanitizers are rather imprecise as they instrument/flag, on average, more than 75% of the functions as problematic, while only 1% of them are actually vulnerable, as shown in Section 7.2 on page 90. Consequently, this large number of false positives prevents sanitizer-guided fuzzing from reaching the full potential of vulnerability-guided fuzzing.

This also applies to the approach of Wang et al. [305], which uses memory-accessing instructions as fuzzing targets. However, the large number of such instructions in real-world C/C++ programs is likely to subsume those instrumented by sanitizers, again leading to imprecise vulnerability-guided fuzzing. This is also reflected in our evaluation, where our SAST-guided fuzzing approach found new security bugs more effectively and efficiently than the aforementioned vulnerability-guided approaches.

Summary

We address Gap 5 (finding a more effective fuzzing strategy) by developing a vulnerability-guided fuzzing approach that (i) uses SAST tool findings as fuzzing targets, (ii) integrates a dynamic target scheduling mechanism to mitigate the impact of likely unreachable and false-positive targets on the fuzzing performance, and (iii) outperforms coverage-based and existing vulnerability-guided approaches in terms of bug-detection effectiveness and efficiency.

9. Conclusion

This chapter concludes this dissertation. It summarizes the main results and insights, discusses overarching limitations, and provides an outlook and ideas for future work.

9.1. Synthesis of Results

We started this dissertation with the objective of developing a novel metric that reflects the efficiency (Goal 1) and effectiveness (Goal 2) of fuzzing more accurately than regular code coverage in the context of C/C++ software. Since code coverage treats all source code as equally important and, therefore, tends to inaccurately assess fuzzing performance, we proposed to refine code coverage by focusing only on the potentially vulnerable code. This involved first investigating techniques for reliably identifying such problematic code (Problem 1), followed by demonstrating the superiority and practicality of our vulnerability-oriented code coverage metric (Problem 2).

Problem 1. We began researching this problem by evaluating the effectiveness of state-of-the-art SAST tools to determine whether such analyzers could reliably identify potentially vulnerable code for our proposed fuzzing metric. In a large-scale evaluation, however, we found that while static analyzers can detect some real-world vulnerabilities, they miss a significant portion of bug-prone code—even when used in combination—and are thus inadequate for our purposes. To address this, we explored the integration of SAST tool findings and code complexity metrics into vulnerability prediction models using ML. Our evaluations showed that the trained models are highly effective at distinguishing between vulnerable and non-vulnerable code, making them a reliable tool for identifying the relevant code for our metric.

Problem 2. The above solution for identifying bug-prone code allowed us to apply our vulnerability-oriented code coverage metric to current fuzzing challenges and evaluate its accuracy in reflecting fuzzing performance relative to regular code coverage.

GOAL 1: To demonstrate our metric's improved (cost-)efficiency, we employed it as a saturation-based stopping criterion for fuzzing campaigns, thereby using a ML vulnerability prediction model to identify the potentially vulnerable code. In a large-scale evaluation, we showed that our stopping criterion achieves significant savings in fuzzing time while rarely missing any security bugs compared to the saturation of triggered program crashes and regular code coverage. Hence, these results confirm the superior accuracy of our metric in reflecting fuzzing (cost-)efficiency.

GOAL 2: To demonstrate our metric's improved effectiveness, we integrated it as an optimization criterion for test generation into a directed fuzzer. Specifically, the generated fuzz inputs target the potentially vulnerable code identified by several SAST tools, thereby using sophisticated mechanisms to mitigate their problem of false positives and negatives. In a large-scale evaluation, we showed that our fuzzing approach finds more and different previously unknown security bugs than coverage-based fuzzing and existing vulnerability-guided approaches. Hence, these results confirm the superior accuracy of our metric in reflecting fuzzing effectiveness.

In conclusion, by utilizing our vulnerability-oriented code coverage metric as a criterion for stopping fuzzing campaigns as well as for generating fuzz inputs, we could confirm its superiority over regular code coverage in reflecting fuzzing (cost-)efficiency and effectiveness, thus fulfilling the overarching objective originally set out for this dissertation. Accordingly, the developed solutions have significant practical relevance, as they decisively contribute to counteracting the high bug-finding costs associated with fuzzing, both from an analytical and a constructive perspective.

9.2. Limitations

Like most research projects, this dissertation is subject to some limitations. While we have discussed specific limitations and threats to the validity of our solution approaches and evaluations in the respective chapters, we summarize the most important overarching limitations below.

Memory-Related Bugs in C/C++ Programs. Since its invention, fuzzing has primarily been used to detect memory-related security bugs in C/C++ software, with many new fuzzers developed in the meantime. This, combined with the fact that C/C++ is still one of the most bug-prone programming languages [248, 293], is why this thesis focuses on C/C++. However, C/C++ no longer seems to be the primary domain of SAST, as many static analyzers now provide more comprehensive and effective support for vulnerabilities specific to other languages, such as JavaScript [3]. While the methodological approaches proposed in this thesis are generally transferable to other languages, the positive results achieved even with less effective C/C++ analyzers suggest that the effectiveness of our solutions is even more pronounced when applied to software written in languages that are better supported by modern SAST tools.

Function-Level Vulnerability Prediction. The lack of datasets with real-world vulnerabilities documented at fine-grained code granularity forced us to use existing datasets with vulnerability information at the function level. For this reason, we also had to evaluate and apply the SAST tools and ML vulnerability prediction models at this level.

For the fuzzing stopping criterion proposed in Chapter 6 on page 73, this meant that our vulnerability-oriented code coverage metric and, thus, the stopping criterion derived from it had to be applied at the function level. Similarly, for the fuzzing approach proposed in

Chapter 7 on page 89, we had to assign the same vulnerability score to all target basic blocks within the same function. Nonetheless, in both cases, we successfully demonstrated[1] the superiority of our fuzzing metric over regular code coverage and existing vulnerability-oriented metrics. Yet, a more fine-grained vulnerability prediction would likely yield even better results, making this an interesting topic for future work once more detailed datasets become available.

Choice of SAST Tools and Subject Programs. Another factor that may limit the insights gained from our evaluations and the effectiveness of our solutions is the selection of SAST tools, subject programs, and the vulnerabilities they contain.

Regarding the choice of SAST tools, apart from our evaluation in Chapter 3 on page 29, we used only freely available analyzers to ensure that our solutions could be applied without requiring costly licenses. However, our evaluation shows that the one commercial SAST tool identifies (potentially) vulnerable code more effectively, so commercial tools would likely further improve the performance of our fuzzing-related solutions.

Our evaluations were also limited to open-source programs as all selected SAST tools and fuzzers require source code access to analyze or instrument the SUT. Although we used various large programs, they may not fully represent proprietary software, which often adheres to stricter security standards enforced by dedicated security teams [146]. Hence, our findings may generalize only to similar open-source programs, leaving their validity on proprietary software uncertain.

With respect to the vulnerabilities used as ground truth in our SAST tool evaluation, one threat is our assumption that the code locations of the root cause provided by the CVEs and those of the manifestations typically flagged by the analyzers generally overlap within the same functions. This poses two issues: (i) the Magma vulnerabilities, for which we observe a high overlap, may introduce a selection bias as they may be easier to detect, and (ii) this overlap may not hold for the vulnerabilities in Binutils and FFmpeg, which lack data for verification. While the former is a threat we cannot exclude, the latter appears to be a reasonable assumption given their similarities to the Magma programs.

Furthermore, static analyzers may have been previously applied to our subject programs. If vulnerabilities had already been identified and fixed earlier during development without being reported as CVEs, the remaining ones would be inherently harder to detect, making the analyzers appear less effective in our evaluation than they actually are. Although we found no clear evidence of prior SAST tool usage on these programs, we cannot completely rule out this possibility.

Automated Deduplication of Crashes Into Unique Bugs. As discussed in the respective chapters, the large number of program crashes triggered during our fuzzing experiments forced us to automate the deduplication of crashes into unique security bugs. This step is crucial to accurately assess the bug-finding (cost-)efficiency and effectiveness of our solutions. We opted for an automated approach because manual deduplication would have required extensive effort and in-depth knowledge of the various subject programs—both of which were hardly feasible within the time constraints of this thesis. Therefore, we

[1] To ensure a fair comparison, the baseline metrics were used at the same code granularity as our metric.

used a well-established deduplication approach (see Section "Number of Unique Bugs" on page 23), which, however, carries the risk of over- or undercounting bugs [144] and thus poses a potential threat to the validity of our results.

Development of Prototypes. All solutions proposed in this thesis have been implemented as software tools, which should be regarded as research prototypes with a strong emphasis on practical applicability. However, they should not be confused with production-ready software in terms of (i) engineering, (ii) applicability, and (iii) usability.

Regarding (i), our tools are largely built on top of other prototypes, inheriting their technical debt, which we did not refactor for time reasons. For the newly developed code, however, we followed modern software engineering practices, including extensive unit testing and profiling of critical parts to ensure their functional correctness and runtime efficiency.

Regarding (ii), the developed software runs only on GNU/Linux-based systems, limiting its use in other environments. To mitigate this dependency (and ensure the reproducibility of our results), we provide Docker containers for all tools and release the corresponding source code as OSS (see Appendix A.1 on page 131).

Regarding (iii), instead of full documentation, we provide instructions for running our tools in the form of README files.

9.3. Outlook and Future Work

This dissertation has touched upon various research areas within the field of SAST and fuzzing, with several directions left open for further exploration. In the following, we discuss the most interesting topics for future work.

Large Language Models to Improve SAST. As mentioned earlier, the accuracy of our vulnerability-oriented code coverage metric depends on the effectiveness of the static and predictive bug detection techniques used to identify potentially vulnerable code. Traditional SAST tools often struggle to fully grasp the semantics of the SUT, resulting not only in numerous wrong analyzer findings but also in missed security bugs, as shown in Chapter 3 on page 29. Large language models (LLMs) [32, 245] may provide a solution here. Recent LLMs have demonstrated remarkable capabilities in program comprehension (e.g., GitHub Copilot) [48, 121, 300], including improving software security [69, 165, 315], making them a potentially valuable complement to SAST tools to address some of their limitations. Therefore, future research could explore the use of LLMs to more reliably identify bug-prone code (i.e., reducing false negatives) or to filter out incorrect SAST tool findings (i.e., reducing false positives). With respect to our metric, advances in either direction would automatically lead to more accurate fuzzing assessments.

Fuzzing of Potentially Vulnerable Code Regressions. Continuing the theme of more effective vulnerability prediction, code regressions offer a promising direction, as demonstrated in several studies [207, 268], particularly by Zhu and Böhme [336] for guiding fuzzing. The underlying assumption is that recently modified source code is more likely to

contain security bugs than long-established code that has already been thoroughly tested. Inspired by this idea, future research could improve our metric by incorporating code regressions. Specifically, instead of analyzing the entire code of the SUT, the employed vulnerability prediction tools could focus only on recent code changes (e.g., provided by the VCS) to identify potentially vulnerable code. This refinement would significantly reduce the amount of code that needs to be considered by our metric, potentially leading to even more accurate assessments of fuzzing (cost-)efficiency and effectiveness.

Incorporating Test Diversity Into Our Metric. Our metric refines regular code coverage by focusing solely on potentially vulnerable code covered during fuzzing. Although we consider the execution frequency of bug-prone code in our vulnerability-guided fuzzing approach (which implements our metric) to fuzz more likely vulnerable code for longer periods, the current version of our metric does not account for the diversity of test data that exercise these locations. This omission can lead to inaccurate fuzzing assessments because vulnerable code can be executed without exposing the underlying security bug if the corresponding vulnerability condition is not satisfied. For instance, incorporating input or output diversity into our metric, as suggested by several works [6, 35, 49, 85], could reduce the risk of overestimation. A higher test diversity (without triggering a program crash) would then increase confidence that the covered code is not vulnerable. Hence, accounting for input diversity would be a challenging but valuable extension of our metric that could be targeted in future work.

Optimizing the Distance Computations of Directed Fuzzing. Another promising direction for future work involves optimizing the distance computations used in directed fuzzing in order to assess the proximity between a fuzz input's execution trace and the targeted code. As observed in our fuzzer evaluation in Chapter 7 on page 89, these computations currently constrain the bug-finding efficiency of our directed fuzzer due to slower test execution and, consequently, lower the throughput of fuzz inputs. This shows that even the most accurate fuzzing metric can lose its effectiveness as a test-generation criterion if its implementation compromises runtime efficiency. This inefficiency is largely due to the fact that currently available directed fuzzers are built on top of existing coverage-based fuzzers, thereby inheriting the technical debt from multiple generations of prototype development. Although various approaches have been introduced to speed up directed fuzzing through selective instrumentation [141, 174] or input filtering [123, 215, 277], a more efficient distance computation implementation remains an open research and engineering problem.

Improving SAST through Fuzzing. While we use SAST in this thesis to improve fuzzing, exploring the reverse, i.e., using fuzzing to improve SAST, presents an interesting and largely unexplored venue for further research. In this context, Murali et al. [203] successfully demonstrated the effectiveness of using fuzzing to prune false positives generated by SAST tools, thereby reducing the manual effort required by developers to review them. However, little research has been done so far on using fuzzing to address false negatives, which is a common problem of SAST tools, as shown in Chapter 3 on page 29. To address

this, the underlying vulnerability patterns in bugs detected by fuzzing but missed by SAST must first be extracted. These patterns must then be translated into security checks for the selected SAST tools. Both steps could be carried out manually or (ideally) automatically, for instance, with the assistance of an LLM. This would allow similar security bugs to be detected earlier in the development process while also reducing the costs associated with fuzzing or direct LLM-based vulnerability prediction.

A. Appendix

A.1. Thesis Artifacts

Below are the links to the GitHub repositories containing the software tools, datasets, and evaluation scripts developed, generated, and used in this thesis, along with detailed descriptions of how to use these artifacts.

Chapter 3: SAST Tool Evaluation

- `https://github.com/tum-i4/sast-tool-evaluation-artifacts`

Chapter 4: ML-Based Vulnerability Prediction

- `https://github.com/tum-i4/green-fuzzing-artifacts`

Chapter 5: Real-Time Coverage Tracking

- `https://github.com/tum-i4/fuzztastic`
- `https://github.com/tum-i4/fuzztastic-dataset`

Chapter 6: Vulnerability-Oriented Fuzzing Stopping Criterion

- `https://github.com/tum-i4/green-fuzzing-artifacts`

Chapter 7: Vulnerability-Guided Fuzzing

- `https://github.com/tum-i4/sast-fuzz`
- `https://github.com/tum-i4/sast-fuzz-artifacts`

Bibliography

[1] Free Software Foundation (FSF). *Gcov: A Test Coverage Program*. Mar. 1, 2024. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[2] Open Source Security Foundation (OpenSSF). *Fuzz-Introspector: Introspect, Extend and Optimise Fuzzers*. Mar. 1, 2024. URL: https://github.com/ossf/fuzz-introspector.

[3] Open Source Security Foundation (OpenSSF). *The OpenSSF CVE Benchmark Project*. Mar. 1, 2024. URL: https://github.com/ossf-cve-benchmark/ossf-cve-benchmark.

[4] Frances E. Allen. "Control Flow Analysis". *ACM SIGPLAN Notices* (1970), pp. 1–19. DOI: 10.1145/390013.808479.

[5] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. "An Empirical Study of Security Warnings From Static Application Security Testing Tools". *Journal of Systems and Software* (2019). DOI: 10.1016/j.jss.2019.110427.

[6] Nadia Alshahwan and Mark Harman. "Augmenting Test Suites Effectiveness by Increasing Output Diversity". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2012, pp. 1345–1348. DOI: 10.1109/ICSE.2012.6227083.

[7] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. "Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study". In: *Proceedings of the European Dependable Computing Conference*. IEEE. 2016, pp. 37–44. DOI: 10.1109/EDCC.2016.34.

[8] Andrea Arcuri and Lionel C. Briand. "A Practical Guide for Using Statistical Tests To Assess Randomized Algorithms in Software Engineering". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2011, pp. 1–10. DOI: 10.1145/1985793.1985795.

[9] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. "Dos and Don'ts of Machine Learning in Computer Security". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2022, pp. 3971–3988.

[10] Andrei Arusoaie, Stefan Ciobaca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code". In: *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE. 2018, pp. 161–168. DOI: 10.1109/SYNASC.2017.00035.

[11] Abhishek Arya and Oliver Chang. *ClusterFuzz: Fuzzing at Google Scale*. Dec. 4, 2019. URL: https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf.

[12] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[13] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. "QL: Object-Oriented Queries on Relational Data". In: *Proceedings of the European Conference on Object-Oriented Programming*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 2016, 2:1–2:25. DOI: 10.4230/LIPIcs.ECOOP.2016.2.

[14] Algirdas Avižienis, Jean Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable and Secure Computing* (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[15] Weiheng Bai, Kefu Wu, Qiushi Wu, and Kangjie Lu. "Guiding Directed Fuzzing With Feasibility". In: *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*. IEEE. 2023, pp. 42–49. DOI: 10.1109/EuroSPW59978.2023.00010.

[16] Vipin Balachandran. "Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2013, pp. 931–940. DOI: 10.1109/ICSE.2013.6606642.

[17] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. "Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software". In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE. 2016, pp. 470–481. DOI: 10.1109/saner.2016.105.

[18] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. "Grammar Mutation for Testing Input Parsers (Registered Report)". In: *Proceedings of the International Fuzzing Workshop*. ACM. 2023, pp. 3–11. DOI: 10.1145/3605157.3605170.

[19] Marcel Böhme. *Guarantees in Software Security*. 2024. arXiv: 2402.01944 [cs.CR].

[20] Marcel Böhme. "STADS: Software Testing as Species Discovery". *ACM Transactions on Software Engineering and Methodology* (2018), pp. 1–52. DOI: 10.1145/3210309.

[21] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections". *IEEE Software* (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[22] Marcel Böhme and Brandon Falk. "Fuzzing: On the Exponential Cost of Vulnerability Discovery". In: *Proceedings of the Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2020, pp. 713–724. DOI: 10.1145/3368089.3409729.

[23] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. "Estimating Residual Risk in Greybox Fuzzing". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2021, pp. 230–241. DOI: 10.1145/3468264.3468570.

[24] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2017, pp. 2329–2344. DOI: 10.1145/3133956.3134020.

[25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2016, pp. 1032–1043. DOI: 10.1145/2976749.2978428.

[26] Marcel Böhme, László Szekeres, and Jonathan Metzman. "On the Reliability of Coverage-Based Fuzzer Benchmarking". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2022, pp. 1621–1633. DOI: 10.1145/3510003.3510230.

[27] Tim Boland and Paul E. Black. "Juliet 1.1 C/C++ and Java Test Suite". *IEEE Computer* (2012), pp. 88–90. DOI: 10.1109/MC.2012.345.

[28] Hudson Borges and Marco Tulio Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform". *Journal of Systems and Software* (2018), pp. 112–129. DOI: 10.1016/j.jss.2018.09.016.

[29] Larissa Braz, Christian Aeberhard, Gul Calikli, and Alberto Bacchelli. "Less Is More: Supporting Developers in Vulnerability Detection During Code Review". In: *Proceedings of the International Conference on Software Engineering*. Association for Computing Machinery. 2022, pp. 1317–1329. DOI: 10.1145/3510003.3511560.

[30] Leo Breiman. "Random Forests". *Machine Learning* (2001), pp. 5–32. DOI: 10.1023/A:1010933404324.

[31] Lionel C. Briand and Dietmar Pfahl. "Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage". In: *Proceedings of the International Symposium on Software Reliability Engineering*. 1999, pp. 148–157. DOI: 10.1109/issre.1999.809319.

[32] Tom B. Brown et al. "Language Models Are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc. 2020, pp. 1877–1901.

[33] Jason Brownlee. *A Tour of Machine Learning Algorithms*. Oct. 11, 2023. URL: https://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/.

[34] Jason Brownlee. *Applied Machine Learning Process*. July 5, 2019. URL: https://machinelearningmastery.com/process-for-working-through-machine-learning-problems/.

[35] Paulo M.S. Bueno, W. Eric Wong, and Mario Jino. "Improving Random Test Sets Using the Diversity Oriented Test Data Generation". In: *Proceedings of the International Workshop on Random Testing*. ACM. 2007, pp. 10–17. DOI: 10.1145/1292414.1292419.

[36] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. "Evaluating Synthetic Bugs". In: *Proceedings of the Asia Conference on Computer and Communications Security*. ACM. 2021, pp. 716–730. DOI: 10.1145/3433210.3453096.

[37] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. "Static Analyzer for Finding Dynamic Programming Errors". *Software: Practice and Experience* (2000), pp. 775–802. DOI: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

[38] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. "Combining Static Analysis Error Traces With Dynamic Symbolic Execution (Experience Paper)". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2022, pp. 568–579. DOI: 10.1145/3533767.3534384.

[39] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. USENIX. 2008, pp. 209–224.

[40] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2006, pp. 322–335. DOI: 10.1145/1180405.1180445.

[41] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". *Communications of the ACM* (2013), pp. 82–90. DOI: 10.1145/2408776.2408795.

[42] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional Shape Analysis by Means of Bi-Abduction". *Journal of the ACM* (2011). DOI: 10.1145/2049697.2049700.

[43] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. "Deep Learning Based Vulnerability Detection: Are We There Yet?" *IEEE Transactions on Software Engineering* (2021), pp. 3280–3296. DOI: 10.1109/TSE.2021.3087402.

[44] Girish Chandrashekar and Ferat Sahin. "A Survey on Feature Selection Methods". *Computers and Electrical Engineering* (2014), pp. 16–28. DOI: 10.1016/j.compeleceng.2013.11.024.

[45] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. *Toward Effective Secure Code Reviews: An Empirical Study of Security-Related Coding Weaknesses*. 2023. arXiv: 2311.16396 [cs.SE].

[46] George Chatzieleftheriou and Panagiotis Katsaros. "Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities". In: *Proceedings of the International Computer Software and Applications Conference*. IEEE. 2011, pp. 96–103. DOI: 10.1109/COMPSACW.2011.26.

[47] Hongxu Chen, Bihuan Chen, Yinxing Xue, Xiaofei Xie, Yang Liu, Yuekang Li, and Xiuheng Wu. "Hawkeye: Towards a Desired Directed Grey-Box Fuzzer". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2018, pp. 2095–2108. DOI: 10.1145/3243734.3243849.

[48] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[49] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. "Adaptive Random Testing: The ART of Test Case Diversity". *Journal of Systems and Software* (2010), pp. 60–66. DOI: 10.1016/j.jss.2009.02.022.

[50] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. "SAVIOR: Towards Bug-Driven Hybrid Testing". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002.

[51] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and Rene Just. "Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size". In: *Proceedings of the International Conference on Automated Software Engineering*. ACM. 2020, pp. 237–249. DOI: 10.1145/3324884.3416667.

[52] Brian Chess and Gary McGraw. "Static Analysis for Security". *IEEE Security and Privacy* (2004), pp. 76–79. DOI: 10.1109/MSP.2004.111.

[53] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. "Grey-Box Concolic Testing on Binary Code". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2019, pp. 736–747. DOI: 10.1109/ICSE.2019.00082.

[54] Nicole Choi. *The Architecture of SAST Tools: An Explainer for Developers*. Feb. 12, 2024. URL: https://github.blog/2024-02-12-the-architecture-of-sast-tools-an-explainer-for-developers/#:~:text=SAST%20tools%20are%20designed%20and,ability%20to%20trace%20data%20flows..

[55] Istehad Chowdhury and Mohammad Zulkernine. "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities". *Journal of Systems Architecture* (2011), pp. 294–313. DOI: 10.1016/j.sysarc.2010.06.003.

[56] Maria Christakis and Christian Bird. "What Developers Want and Need From Program Analysis: An Empirical Study". In: *Proceedings of the International Conference on Automated Software Engineering*. ACM. 2016, pp. 332–343. DOI: 10.1145/2970276.2970347.

[57] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. "On the Number and Nature of Faults Found by Random Testing". *Software Testing, Verification and Reliability* (2011), pp. 3–28. DOI: 10.1002/stvr.415.

[58] Lori A. Clarke. "A System To Generate Test Data and Symbolically Execute Programs". *IEEE Transactions on Software Engineering* (1976), pp. 215–222. DOI: 10.1109/TSE.1976.233817.

[59] OASIS SARIF Technical Committee. *Static Analysis Results Interchange Format (SARIF) Version 2.1.0*. Oct. 1, 2024. URL: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html.

[60] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". *Machine Learning* (1995), pp. 273–297. DOI: 10.1023/A:1022627411411.

[61] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the Symposium on Principles of Programming Languages*. ACM. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[62] Roland Croft, Dominic Newlands, Ziyu Chen, and Ali M. Babar. "An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing". In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM. 2021. DOI: 10.1145/3475716.3475781.

[63] Ivan Bjerre Damgård. "Collision Free Hash Functions and Public Key Signature Schemes". In: *Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*. Springer. 1987, pp. 203–216.

[64] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2012, pp. 1084–1093.

[65] Jared D. DeMott, Richard J. Enbody, and William F. Punch. "Revolutionizing the Field of Grey-Box Attack Surface Testing With Evolutionary Fuzzing". *BlackHat and DefCon* (2007).

[66] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow". *Communications of the ACM* (1977), pp. 504–513. DOI: 10.1145/359636.359712.

[67] Edsger W. Dijkstra. "A Note on Two Problems in Connexion With Graphs". *Numerische Mathematik* (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[68] Edsger W. Dijkstra. *Notes on Structured Programming (EWD249)*. Tech. rep. Technological University Eindhoven, 1970.

[69] Dinil Mon Divakaran and Sai Teja Peddinti. *LLMs for Cyber Security: New Opportunities*. 2024. arXiv: 2404.11338 [cs.CR].

[70] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. "LAVA: Large-Scale Automated Vulnerability Addition". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[71] Pedro Domingos. "A Few Useful Things To Know About Machine Learning". *Communications of the ACM* (2012), pp. 78–87. DOI: 10.1145/2347736.2347755.

[72] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. "Microservices: Yesterday, Today, and Tomorrow". *Present and Ulterior Software Engineering* (2017), pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[73] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. "Leopard: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2019, pp. 60–71. DOI: 10.1109/ICSE.2019.00024.

[74] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. "WindRanger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2022, pp. 2440–2451. DOI: 10.1145/3510003.3510197.

[75] Zakir Durumeric et al. "The Matter of Heartbleed". In: *Proceedings of the Internet Measurement Conference*. 2014. DOI: 10.1145/2663716.2663755.

[76] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. First. Addison-Wesley, 1999.

[77] Russell Eberhart and James Kennedy. "New Optimizer Using Particle Swarm Theory". In: *Proceedings of the International Symposium on Micro Machine and Human Science*. IEEE. 1995, pp. 39–43. DOI: 10.1109/mhs.1995.494215.

[78] Karim O. Elish and Mahmoud O. Elish. "Predicting Defect-Prone Software Modules Using Support Vector Machines". *Journal of Systems and Software* (2008), pp. 649–660. DOI: 10.1016/j.jss.2007.07.040.

[79] Ericsson. *CodeChecker: A Static Analysis Infrastructure Built on the LLVM/Clang Static Analyzer Toolchain*. Mar. 1, 2024. URL: https://codechecker.readthedocs.io/en/latest/.

[80] David Evans and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". *IEEE Software* (2002), pp. 42–51. DOI: 10.1109/52.976940.

[81] Jerome Fan, Suneel Upadhye, and Andrew Worster. "Understanding Receiver Operating Characteristic (ROC) Curves". *Canadian Journal of Emergency Medicine* (2006), pp. 19–20. DOI: 10.1017/S1481803500013336.

[82] Anum Fatima, Shazia Bibi, and Rida Hanif. "Comparative Study on Static Code Analysis Tools for C/C++". In: *Proceedings of the International Bhurban Conference on Applied Sciences and Technology*. IEEE. 2018, pp. 465–469. DOI: 10.1109/IBCAST.2018.8312265.

[83] Ansgar Fehnker, Ralf Huuck, Sean Seefried, and Michael Tapp. "Fade to Grey: Tuning Static Program Analysis". *Electronic Notes in Theoretical Computer Science* (2010), pp. 17–32. DOI: 10.1016/j.entcs.2010.08.046.

[84] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. "Security Testing: A Survey". In: *Advances in Computers*. Elsevier, 2016, pp. 1–51. DOI: 10.1016/bs.adcom.2015.11.003.

[85] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. "Searching for Cognitively Diverse Tests: Towards Universal Test Diversity Metrics". In: *International Conference on Software Testing Verification and Validation Workshop*. IEEE. 2008, pp. 178–186. DOI: 10.1109/ICSTW.2008.36.

[86] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *Proceedings of the USENIX Workshop on Offensive Technologies*. USENIX. 2020.

[87] Phyllis G. Frankl and J. Weyuker. "Provable Improvements on Branch Testing". *IEEE Transactions on Software Engineering* (1993), pp. 962–975. DOI: 10.1109/32.245738.

[88] Jerome H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine". *Annals of Statistics* (2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451.

[89] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. "CollAFL: Path Sensitive Fuzzing". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2018, pp. 679–696. DOI: 10.1109/SP.2018.00040.

[90] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. "Choosing Software Metrics for Defect Prediction: An Investigation on Feature Selection Techniques". *Software: Practice and Experience* (2011), pp. 579–606. DOI: 10.1002/spe.1043.

[91] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin I.P. Rubinstein. "Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report)". In: *Proceedings of the International Fuzzing Workshop*. ACM. 2023, pp. 47–55. DOI: 10.1145/3605157.3605177.

[92] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. "Prioritizing Software Security Fortification Through Code-Level Metrics". In: *Proceedings of the 4th ACM Workshop on Quality of Protection*. ACM. 2008, pp. 31–38. DOI: 10.1145/1456362.1456370.

[93] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. *An Empirical Study on Benchmarks of Artificial Software Vulnerabilities*. 2020. arXiv: 2003.09561 [cs.CR].

[94] Christoph Gentsch. *Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C*. Tech. rep. German Aerospace Center (DLR DW), 2020.

[95] GitHub. *CodeQL: Discover Vulnerabilities Across a Codebase With CodeQL, Our Industry-Leading Semantic Code Analysis Engine*. Mar. 1, 2024. URL: https://codeql.github.com/.

[96] Patrice Godefroid. "Fuzzing: Hack, Art, and Science". *Communications of the ACM* (2020), pp. 70–76. DOI: 10.1145/3363824.

[97] Patrice Godefroid, Michael Y. Levin, and David a. Molnar. "Automated Whitebox Fuzz Testing". In: *Proceedings of the Network and Distributed System Security Symposium*. Internet Society. 2008, pp. 151–166.

[98] John B. Goodenough and Susan L. Gerhart. "Toward a Theory of Test Data Selection". *ACM SIGPLAN Notices* (1975), pp. 493–510. DOI: 10.1145/390016.808473.

[99] Google. *Honggfuzz: Security Oriented Software Fuzzer*. Jan. 1, 2024. URL: https://honggfuzz.dev/.

[100] Google. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Jan. 1, 2024. URL: https://google.github.io/oss-fuzz/.

[101] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code Coverage for Suite Evaluation by Developers". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2014, pp. 72–82. DOI: 10.1145/2568225.2568278.

[102] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the Capability of Static Code Analysis To Detect Security Vulnerabilities". *Information and Software Technology* (2015), pp. 18–33. DOI: 10.1016/j.infsof.2015.08.002.

[103] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. "Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities". *IEEE Transactions on Software Engineering* (2023), pp. 5154–5188. DOI: 10.1109/TSE.2023.3329667.

[104] Isabelle Guyon and André Elisseeff. "An Introduction to Variable and Feature Selection". *Journal of Machine Learning Research* (2003), pp. 1157–1182.

[105] Andrew Habib and Michael Pradel. "How Many of All Bugs Do We Find? A Study of Static Bug Detectors". In: *Proceedings of the International Conference on Automated Software Engineering*. ACM. 2018, pp. 317–328. DOI: 10.1145/3238147.3238213.

[106] Aviad Hahami. *An Unintimidating Introduction to the Dark Arts of C/C++ Vulnerabilities*. Apr. 15, 2022. URL: https://snyk.io/blog/unintimidating-intro-to-c-cpp-vulnerabilities/.

[107] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2013, pp. 49–64.

[108] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. First. Elsevier, 1977.

[109] Richard Hamlet. "Random Testing". *Encyclopedia of Software Engineering* (1994), pp. 971–978.

[110] Keno Hassler, Philipp Görz, Stephan Lipp, Thorsten Holz, and Marcel Böhme. "A Comparative Study of Fuzzers and Static Analysis Tools for Finding Memory Unsafety". *ACM Transactions on Software Engineering and Methodology* (2025). Under submission.

[111] Red Hat. *Security in the software development lifecycle*. Sept. 30, 2022. URL: https://www.redhat.com/en/topics/security/software-development-lifecycle-security.

[112] Ben Hawkes. *Robots Dream of Root Shells*. Mar. 11, 2024. URL: https://blog.isosceles.com/robots-dream-of-root-shells/.

[113] Ben Hawkes. *The WebP 0day*. Sept. 21, 2023. URL: https://blog.isosceles.com/the-webp-0day/.

[114] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2021), pp. 1–29. DOI: 10.1145/3428334.

[115] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. "An Empirical Study on Software Defect Prediction With a Simplified Metric Set". *Information and Software Technology* (2015), pp. 170–190. DOI: 10.1016/j.infsof.2014.11.006.

[116] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. "An Investigation on the Feasibility of Cross-Project Defect Prediction". *Automated Software Engineering* (2012), pp. 167–199. DOI: 10.1007/s10515-011-0090-3.

[117] Mark Hermeling. *Speeding up SAST*. Nov. 4, 2022. URL: https://codesecure.com/learn/speeding-up-sast/.

[118] Adrian Herrera, Mathias Payer, and Antony L. Hosking. "DatAFLow: Toward a Data-Flow-Guided Fuzzer". *ACM Transactions on Software Engineering and Methodology* (2023). DOI: 10.1145/3587156.

[119] Gerard J. Holzmann. "UNO: Static Source Code Checking for User-Defined Properties". In: *Proceedings of the World Conference on Integrated Design and Process Technology*. Society for Design and Process Science. 2002.

[120] David W. Hosmer, Stanley Lemeshow, and Rodney X. Sturdivant. *Applied Logistic Regression*. Third. John Wiley & Sons, 2013.

[121] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. *Large Language Models for Software Engineering: A Systematic Literature Review*. 2024. arXiv: 2308.10620 [cs.SE].

[122] William E. Howden. "Methodology for the Generation of Program Test Data". *IEEE Transactions on Computers* (1975), pp. 554–560. DOI: 10.1109/T-C.1975.224259.

[123] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. "BEACON: Directed Grey-Box Fuzzing With Provable Path Pruning". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2022, pp. 36–50. DOI: 10.1109/SP46214.2022.9833751.

[124] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. "Titan: Efficient Multi-Target Directed Greybox Fuzzing". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2024. DOI: 10.1109/SP54263.2024.00059.

[125] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. "Everything Is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2024. DOI: 10.1109/SP54263.2024.00142.

[126] Lorenz Hüther, Karsten Sohr, Bernhard J. Berger, Hendrik Rothe, and Stefan Edelkamp. "Machine Learning for SAST: A Lightweight and Adaptable Approach". In: *Proceedings of the European Symposium on Research in Computer Security*. Springer. 2023, pp. 85–104.

[127] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. "Challenges With Responding to Static Analysis Tool Alerts". In: *Proceedings of the International Working Conference on Mining Software Repositories*. IEEE. 2019, pp. 245–249. DOI: 10.1109/MSR.2019.00049.

[128] Vector Informatik. *PC-lint Plus: Static Code Analysis Tool for C and C++ Source Code*. Mar. 1, 2024. URL: https://pclintplus.com/.

[129] Laura Inozemtseva and Reid Holmes. "Coverage Is Not Strongly Correlated With Test Suite Effectiveness". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2014, pp. 435–445. DOI: 10.1145/2568225.2568271.

[130] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. "Code Coverage at Google". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2019, pp. 955–963. DOI: 10.1145/3338906.3340459.

[131] Gul Jabeen, Sabit Rahim, Wasif Afzal, Dawar Khan, Aftab Ahmed Khan, Zahid Hussain, and Tehmina Bibi. "Machine Learning Techniques for Software Vulnerability Prediction: A Comparative Study". *Applied Intelligence* (2022), pp. 17614–17635. DOI: 10.1007/s10489-022-03350-5.

[132] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. "TIFF: Using Input Type Inference To Improve Fuzzing". In: *Proceedings of the Annual Computer Security Applications Conference*. ACM. 2018, pp. 505–517. DOI: 10.1145/3274694.3274746.

[133] Tiantian Ji, Zhongru Wang, Zhihong Tian, Binxing Fang, Qiang Ruan, Haichen Wang, and Wei Shi. "AFLPro: Direction Sensitive Fuzzing". *Journal of Information Security and Applications* (2020). DOI: 10.1016/j.jisa.2020.102497.

[134] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. "Dictionary Learning Based Software Defect Prediction". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2014, pp. 414–423. DOI: 10.1145/2568225.2568320.

[135] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[136] Stephen C. Johnson. *Lint, a C Program Checker*. Tech. rep. Bell Laboratories, 1978.

[137] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. "Taming False Alarms From a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis". In: *Proceedings of the International Conference on Static Analysis*. Springer. 2005, pp. 203–217. DOI: 10.1007/11547662_15.

[138] Arvinder Kaur and Ruchikaa Nayyar. "A Comparative Study of Static Code Analysis Tools for Vulnerability Detection in C/C++ and Java Source Code". *Procedia Computer Science* (2020), pp. 2023–2029. DOI: 10.1016/j.procs.2020.04.217.

[139] Sunghun Kim and Michael D. Ernst. "Which Warnings Should I Fix First?" In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM. 2007, pp. 45–54. DOI: 10.1145/1287624.1287633.

[140] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead. "Automatic Identification of Bug-Introducing Changes". In: *Proceedings of the International Conference on Automated Software Engineering*. IEEE. 2006, pp. 81–90. DOI: 10.1109/ASE.2006.23.

[141] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. "DAFL: Directed Grey-Box Fuzzing Guided by Data Dependency". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2023, pp. 4931–4948.

[142] Yong Woo Kim. "Efficient Use of Code Coverage in Large-Scale Software Development". In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. IBM. 2003, pp. 145–155.

[143] James C. King. "Symbolic Execution and Program Testing". *Communications of the ACM* (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[144] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804.

[145] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. First. O'Reilly, 2017.

[146] Nimrod Kramer. *Open Source vs Proprietary Software: Security Comparison*. May 10, 2024. URL: https://daily.dev/de/blog/open-source-vs-proprietary-software-security-comparison.

[147] Jan Kratochvil. *Memory Error Checking In C and C++: Comparing Sanitizers and Valgrind*. May 5, 2021. URL: https://developers.redhat.com/blog/2021/05/05/memory-error-checking-in-c-and-c-comparing-sanitizers-and-valgrind#outofboundsglobal.

[148] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. "Discovering Software Vulnerabilities Using Data-Flow Analysis and Machine Learning". In: *Proceedings of the International Conference on Availability, Reliability and Security*. ACM. 2018, pp. 1–10. DOI: 10.1145/3230833.3230856.

[149] Max Kuhn. "Building Predictive Models in R Using the Caret Package". *Journal of Statistical Software* (2008), pp. 1–26. DOI: 10.18637/jss.v028.i05.

[150] Frans Kunst. *Lint, a C Program Checker*. Tech. rep. Vrije Universiteit Amsterdam, 1988.

[151] William Landi. "Undecidability of Static Analysis". *ACM Letters on Programming Languages and Systems* (1992), pp. 323–337. DOI: 10.1145/161494.161501.

[152] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. "Righting Software". *IEEE Software* (2004), pp. 92–100. DOI: 10.1109/MS.2004.1293079.

[153] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[154] Gwangmu Lee and Byoungyoung Lee. "Constraint-Guided Directed Greybox Fuzzing". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2021, pp. 3559–3576.

[155] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. "Micro Interaction Metrics for Defect Prediction". In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM. 2011, pp. 311–321. DOI: 10.1145/2025113.2025156.

[156] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. "PerfFuzz: Automatically Generating Pathological Inputs". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2018, pp. 254–265. DOI: 10.1145/3213846.3213874.

[157] Caroline Lemieux and Koushik Sen. "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage". In: *Proceedings of the International Conference on Automated Software Engineering*. ACM. 2018, pp. 475–485. DOI: 10.1145/3238147.3238176.

[158] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. "UpSet: Visualization of Intersecting Sets". *IEEE Transactions on Visualization and Computer Graphics* (2014), pp. 1983–1992. DOI: 10.1109/TVCG.2014.2346248.

[159] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. "Software Defect Prediction via Convolutional Neural Network". In: *Proceedings of the International Conference on Software Quality, Reliability and Security*. IEEE. 2017, pp. 318–328. DOI: 10.1109/QRS.2017.42.

[160] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java". In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2023, pp. 921–933. DOI: 10.1145/3611643.3616262.

[161] Penghui Li, Yinxi Liu, and Wei Meng. "Understanding and Detecting Performance Bugs in Markdown Compilers". In: *Proceedings of the International Conference on Automated Software Engineering*. IEEE. 2021, pp. 892–904. DOI: 10.1109/ASE51524.2021.9678611.

[162] Rundong Li, HongLiang Liang, Liming Liu, Xutong Ma, Rong Qu, Jun Yan, and Jian Zhang. "GTFuzz: Guard Token Directed Grey-Box Fuzzing". In: *Proceedings of the Pacific Rim International Symposium on Dependable Computing*. IEEE. 2020, pp. 160–170. DOI: 10.1109/PRDC50213.2020.00027.

[163] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. "V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs". *IEEE Transactions on Cybernetics* (2020), pp. 1–12. DOI: 10.1109/TCYB.2020.3013675.

[164] Yuwei Li et al. "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2021, pp. 2777–2794.

[165] Ziyang Li, Saikat Dutta, and Mayur Naik. *LLM-Assisted Static Analysis for Detecting Security Vulnerabilities*. 2024. arXiv: 2405.17238 [cs.CR].

[166] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. "Sequence Directed Hybrid Fuzzing". In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*. IEEE. 2020, pp. 127–137. DOI: 10.1109/SANER48275.2020.9054807.

[167] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. "Sequence Coverage Directed Greybox Fuzzing". In: *Proceedings of the International Conference on Program Comprehension*. IEEE. 2019, pp. 249–259. DOI: 10.1109/ICPC.2019.00044.

[168] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection". In: *Proceedings of the International Symposium on Software Testing and Analysis*. Copyright: ACM. ACM. 2022, pp. 544–555. DOI: 10.1145/3533767.3534380.

[169] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. "FuzzTastic: A Fine-Grained, Fuzzer-Agnostic Coverage Analyzer". In: *Companion Proceedings of the International Conference on Software Engineering*. Copyright: Owner/Author (CC-BY-4.0). ACM. 2022, pp. 75–79. DOI: 10.1145/3510454.3516847.

[170] Stephan Lipp, Daniel Elsner, Severin Kacianka, Alexander Pretschner, Marcel Böhme, and Sebastian Banescu. "Green Fuzzing: A Saturation-Based Stopping Criterion Using Vulnerability Prediction". In: *Proceedings of the International Symposium on Software Testing and Analysis*. Copyright: ACM. ACM. 2023, pp. 127–139. DOI: 10.1145/3597926.3598043.

[171] Stephan Lipp, Severin Kacianka, Alexander Pretschner, and Marcel Böhme. "SAST-Guided Greybox Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. Under submission. ACM. 2026.

[172] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, Chao Zhang, and Chao Zhang. "A Large-Scale Empirical Study on Vulnerability Distribution Within Projects and the Lessons Learned". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2020, pp. 1547–1559. DOI: 10.1145/3377811.3380923.

[173] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. "BugBench: Benchmarks for Evaluating Bug Detection Tools". In: *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. unknown. 2005, pp. 1–5.

[174] Changhua Luo, Wei Meng, and Penghui Li. "SelectFuzz: Efficient Directed Fuzzing With Selective Path Exploration". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2023, pp. 2693–2707. DOI: 10.1109/SP46215.2023.10179296.

[175] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei Han Lee, Yu Song, and Raheem Beyah. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2019, pp. 1949–1966.

[176] Rupak Majumdar and Koushik Sen. "Hybrid Concolic Testing". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[177] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, sang kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". *IEEE Transactions on Software Engineering* (2019), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[178] Paul Marinescu. *Autonomous testing of services at scale*. Oct. 20, 2021. URL: https://engineering.fb.com/2021/10/20/developer-tools/autonomous-testing/.

[179] Daniel Marjamäki. *Cppcheck: A Tool for Static C/C++ Code Analysis*. Mar. 1, 2024. URL: https://cppcheck.sourceforge.io/.

[180] Wes Masri. "Automated Fault Localization: Advances and Challenges". In: Elsevier, 2015, pp. 103–156. DOI: 10.1016/bs.adcom.2015.05.001.

[181] MathWorks. *PolySpace: Test Software and Assess Code Quality*. Mar. 1, 2024. URL: https://de.mathworks.com/products/polyspace.html.

[182] Dastan Maulud and Adnan M. Abdulazeez. "A Review on Linear Regression Comprehensive in Machine Learning". *Journal of Applied Science and Technology Trends* (2020), pp. 140–147. DOI: 10.38094/jastt1457.

[183] Thomas J. McCabe. "A Complexity Measure". *IEEE Transactions on Software Engineering* (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[184] Gary McGraw. "Software Security". *IEEE Security and Privacy* (2004), pp. 80–83. DOI: 10.1109/MSECP.2004.1281254.

[185] Nadia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. "Vulnerable Code Detection Using Software Metrics and Machine Learning". *IEEE Access* (2020), pp. 219174–219198. DOI: 10.1109/ACCESS.2020.3041181.

[186] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. "Fitness Guided Vulnerability Detection With Greybox Fuzzing". In: *Proceedings of the International Conference on Software Engineering Workshops*. ACM. 2020, pp. 513–520. DOI: 10.1145/3387940.3391457.

[187] Dongyu Meng, Michele Guerriero, Aravind MacHiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. "Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms". In: *Proceedings of the Asia Conference on Computer and Communications Security*. ACM. 2021, pp. 731–743. DOI: 10.1145/3433210.3453115.

[188] Meta. *Infer: A Tool To Detect Bugs in Java and C/C++/Objective-C Code Before It Ships*. Mar. 1, 2024. URL: https://fbinfer.com/.

[189] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. "FuzzBench: An Open Fuzzer Benchmarking Platform and Service". In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2021, pp. 1393–1403. DOI: 10.1145/3468264.3473932.

[190] Microsoft. *OneFuzz: A Self-Hosted Fuzzing-as-a-Service Platform*. Jan. 21, 2024. URL: https://github.com/microsoft/onefuzz.

[191] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". *Communications of the ACM* (1990), pp. 32–44. DOI: 10.1145/96267.96279.

[192] Charlie Miller. *Fuzz by Number: More Data About Fuzzing Than You Ever Wanted To Know*. Mar. 28, 2008. URL: https://www.ise.io/wp-content/uploads/2019/11/cmiller_cansecwest2008.pdf.

[193] MITRE. *2023 CWE Top 25 Most Dangerous Software Weaknesses*. May 1, 2024. URL: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[194] MITRE. *Common Weakness Enumeration (CWE): A Community-Developed List of Software and Hardware Weaknesses That Can Become Vulnerabilities*. Mar. 1, 2024. URL: https://cwe.mitre.org/.

[195] Anders Møller and Michael I. Schwartzbach. "Static Program Analysis". *Lecture Notes* (2024).

[196] Max Moroz and Kostya Serebryany. *Guided in-process fuzzing of Chrome components*. Aug. 5, 2016. URL: https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html.

[197] Ben Moseley and Peter Marks. "Out of the Tar Pit". In: *Proceedings of the Software Practice Advancement Conference*. 2006.

[198] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2008, pp. 181–190. DOI: 10.1145/1368088.1368114.

[199] Sara Moshtari and Ashkan Sami. "Evaluating and Comparing Complexity, Coupling and a New Proposed Set of Coupling Metrics in Cross-Project Vulnerability Prediction". In: *Proceedings of the ACM Symposium on Applied Computing*. ACM. 2016, pp. 1415–1421. DOI: 10.1145/2851613.2851777.

[200] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. "Using Complexity Metrics To Improve Software Security". *Computer Fraud and Security* (2013), pp. 8–17. DOI: 10.1016/S1361-3723(13)70045-9.

[201] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: Introduction and Applications". *Communications of the ACM* (2011), pp. 69–77. DOI: 10.1145/1995376.1995394.

[202] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. "Understanding the Reproducibility of Crowd-Reported Security Vulnerabilities". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2018, pp. 919–936.

[203] Aniruddhan Murali, Noble S. Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. "FuzzSlice: Pruning False Positives in Static Analysis Warnings Through Function-Level Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2024, pp. 1–13. DOI: 10.1145/3597503.3623321.

[204] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. "Repositioning of Static Analysis Alarms". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2018, pp. 187–197. DOI: 10.1145/3213846.3213850.

[205] Nachiappan Nagappan and Thomas Ball. "Static Analysis Tools as Early Indicators of Pre-Release Defect Density". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2005, pp. 580–586. DOI: 10.1145/1062455.1062558.

[206] Nachiappan Nagappan and Thomas Ball. "Use of Relative Code Churn Measures To Predict System Defect Density". In: *Proceedings International Conference on Software Engineering*. IEEE. 2005, pp. 284–292. DOI: 10.1109/ICSE.2005.1553571.

[207] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. "Change Bursts as Defect Predictors". In: *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE. 2010, pp. 309–318. DOI: 10.1109/ISSRE.2010.25.

[208] Jaechang Nam and Sunghun Kim. "CLAMI: Defect Prediction on Unlabeled Datasets". In: *Proceedings of the International Conference on Automated Software Engineering*. IEEE. 2015, pp. 452–463. DOI: 10.1109/ASE.2015.56.

[209] Office of the National Cyber Director (ONCD). *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Tech. rep. The White House, 2024.

[210] John A. Nelder and Robert W.M. Wedderburn. "Generalized Linear Models". *Journal of the Royal Statistical Society Series A: Statistics in Society* (1972), pp. 370–384.

[211] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. "Predicting Vulnerable Software Components". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2007, pp. 529–540. DOI: 10.1145/1315245.1315311.

[212] Hoang Lam Nguyen and Lars Grunske. "BeDivFuzz: Integrating Behavioral Diversity Into Generator-Based Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2022, pp. 249–261. DOI: 10.1145/3510003.3510182.

[213] Manh Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. "Binary-Level Directed Fuzzing for Use-After-Free Vulnerabilities". In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. USENIX. 2020, pp. 47–62.

[214] Viet Hung Nguyen and Le Minh Sang Tran. "Predicting Vulnerable Software Components With Dependency Graphs". In: *Proceedings of the International Workshop on Security Measurements and Metrics*. ACM. 2010, pp. 1–8. DOI: 10.1145/1853919.1853923.

[215] Chaitra Niddodi, Stefan Nagy, Darko Marinov, and Sibin Mohan. *TOPr: Enhanced Static Code Pruning for Fast and Precise Directed Fuzzing*. 2023. arXiv: 2309.09522 [cs.SE].

[216] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. "Open Science in Software Engineering: A Study on Deep Learning-Based Vulnerability Detection". *IEEE Transactions on Software Engineering* (2023), pp. 1983–2005. DOI: 10.1109/TSE.2022.3207149.

[217] James R. Norris. *Markov Chains*. First. Cambridge University Press, 1998.

[218] Saahil Ognawala. "Scalable Greybox Fuzzing for Effective Vulnerability Management". PhD thesis. Technical University of Munich, 2020.

[219] Vadim Okun, Aurelien Delaitre, and Paul E. Black. "Report on the Static Analysis Tool Exposition (SATE) IV". *NIST Special Publication* (2011). DOI: 10.6028/NIST.SP.500-297.

[220] Emmanuel Oni. *Safeguarding Software Quality: Tackling False Negatives with Security by Design*. Aug. 29, 2023. URL: https://www.securitycompass.com/blog/safeguarding-software-quality-tackling-false-negatives-with-security-by-design/.

[221] Alfrick Opidi. *How To Address SAST False Positives in Application Security Testing*. Mar. 10, 2022. URL: https://www.mend.io/blog/sast-false-positives/.

[222] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "ParmeSan: Sanitizer-Guided Greybox Fuzzing". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2020, pp. 2289–2306.

[223] Jiradet Ounjai, Valentin Wüstholz, and Maria Christakis. "Green Fuzzer Benchmarking". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2023, pp. 1396–1406. DOI: 10.1145/3597926.3598144.

[224] Brian S. Pak. "Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution". MA thesis. Carnegie Mellon University, 2012.

[225] Sushant K. Pandey, Ravi B. Mishra, and Anil K. Tripathi. "Machine Learning Based Methods for Software Fault Prediction: A Survey". *Expert Systems with Applications* (2021). DOI: 10.1016/j.eswa.2021.114595.

[226] David Lorge Parnas, Paul C. Clements, and David M. Weiss. "The Modular Structure of Complex Systems". *IEEE Transactions on Software Engineering* (1985). DOI: 10.1109/TSE.1985.232209.

[227] Steve Pate. *Measuring the Aftershocks of Heartbleed*. May 13, 2014. URL: https://www.securitymagazine.com/articles/85506-measuring-the-aftershocks-of-heartbleed.

[228] Mathias Payer. "The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes". *IEEE Security and Privacy Magazine* (2019), pp. 78–82. DOI: 10.1109/MSEC.2018.2889892.

[229] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. "1dVul: Discovering 1-Day Vulnerabilities Through Binary Patches". In: *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE. 2019, pp. 605–616. DOI: 10.1109/DSN.2019.00066.

[230] José D'Abruzzo Pereira, Joao R. Campos, and Marco Vieira. "Machine Learning To Combine Static Analysis Alerts With Software Metrics To Detect Security Vulnerabilities: An Empirical Study". In: *Proceedings of the European Dependable Computing Conference*. IEEE. 2021, pp. 1–8. DOI: 10.1109/EDCC53658.2021.00008.

[231] José D'Abruzzo Pereira and Marco Vieira. "On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects". In: *Proceedings of the European Dependable Computing Conference*. IEEE. 2020, pp. 97–102. DOI: 10.1109/EDCC51268.2020.00025.

[232] Ani Petrosyan. *Estimated Cost of Cybercrime Worldwide 2017–2028*. Nov. 15, 2023. URL: https://www.statista.com/forecasts/1280009/cost-cybercrime-worldwide.

[233] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2017, pp. 2155–2168. DOI: 10.1145/3133956.3134073.

[234] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. First. Wiley, 2008.

[235] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. "Smart Greybox Fuzzing". *IEEE Transactions on Software Engineering* (2019), pp. 1980–1997. DOI: 10.1109/TSE.2019.2941681.

[236] Paul Piwowarski. "A Nesting Level Complexity Measure". *ACM Sigplan Notices* (1982), pp. 44–50. DOI: 10.1145/947955.947960.

[237] LLVM Project. *Clang Static Analyzer*. Mar. 1, 2024. URL: https://clang-analyzer.llvm.org/.

[238] LLVM Project. *Clang-Tidy: Extra Clang Tools*. Mar. 1, 2024. URL: https://clang.llvm.org/extra/clang-tidy/.

[239] LLVM Project. *Clang: Source-Based Code Coverage*. Jan. 21, 2024. URL: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[240] LLVM Project. *LibFuzzer: A Library for Coverage-Guided Fuzz Testing*. Jan. 1, 2024. URL: https://llvm.org/docs/LibFuzzer.html.

[241] LLVM Project. *Writing an LLVM Pass*. May 1, 2024. URL: https://llvm.org/docs/WritingAnLLVMNewPMPass.html.

[242] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Ding Yang, Shun Li, Binyu Li, and Zhenyu Chen. "DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing". *ACM Transactions on Software Engineering and Methodology* (2024). DOI: 10.1145/3654440.

[243] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. "Deep Learning Based Software Defect Prediction". *Neurocomputing* (2020), pp. 100–110. DOI: 10.1016/j.neucom.2019.11.067.

[244] J. R. Quinlan. "Induction of Decision Trees". *Machine Learning* (1986), pp. 81–106. DOI: 10.1023/A:1022643204877.

[245] Alec Radfort, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. *Improving Language Understanding by Generative Pre-Training*. Tech. rep. OpenAI, 2018.

[246] Foyzur Rahman and Premkumar Devanbu. "How, and Why, Process Metrics Are Better". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2013, pp. 432–441. DOI: 10.1109/ICSE.2013.6606589.

[247] Chitoor V. Ramamoorthy, Siu Bun F. Ho, and W. T. Chen. "On the Automated Generation of Program Test Data". *IEEE Transactions on Software Engineering* (1976), pp. 293–300. DOI: 10.1109/TSE.1976.233835.

[248] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. "A Large Scale Study of Programming Languages and Code Quality in GitHub". In: *Proceedings of the Symposium on the Foundations of Software Engineering*. 2014. DOI: 10.1145/2635868.2635922.

[249] Reasoning. *Illuma*. Mar. 1, 2024. URL: http://www.reasoning.com/.

[250] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74. DOI: 10.1109/lics.2002.1029817.

[251] Eric F. Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. "On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2016, pp. 132–143. DOI: 10.1145/2884781.2884835.

[252] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". *Nature* (1986), pp. 533–536. DOI: 10.1038/323533a0.

[253] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *Proceedings of the International Conference on Machine Learning and Applications*. IEEE. 2019, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[254] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2008, pp. 341–350. DOI: 10.1145/1368088.1368135.

[255] Andrei Sabelfeld and Andrew C. Myers. "Language-Based Information-Flow Security". *IEEE Journal on Selected Areas in Communications* (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[256] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "Lessons From Building Static Analysis Tools at Google". *Communications of the ACM* (2018), pp. 58–66. DOI: 10.1145/3188720.

[257] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. "Predicting Vulnerable Software Components via Text Mining". *IEEE Transactions on Software Engineering* (2014), pp. 993–1006. DOI: 10.1109/TSE.2014.2340398.

[258] Moritz Schlögel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. "SoK: Prudent Evaluation Practices for Fuzzing". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2024. DOI: 10.1109/SP54263.2024.00137.

[259] Patrick Schober, Christa Boer, and Lothar A. Schwarte. "Correlation Coefficients: Appropriate Use and Interpretation". *Anesthesia and Analgesia* (2018), pp. 1763–1768. DOI: 10.1213/ANE.0000000000002864.

[260] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted To Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid To Ask)". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[261] Robert C Seacord. *Secure Coding in C and C++*. Second. Addison-Wesley, 2013.

[262] Semgrep. *Semgrep: Find Bugs and Enforce Code Standards*. Mar. 1, 2024. URL: https://semgrep.dev/.

[263] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the USENIX Annual Technical Conference*. USENIX. 2019, pp. 309–318.

[264] Kostya Serebryany. *OSS-Fuzz: Google's Continuous Fuzzing Service for Open-Source Software*. Publisher: USENIX. 2017. URL: https://www.usenix.org/sites/default/files/conference/protected-files/usenixsecurity17_slides_serebryany.pdf.

[265] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. "MC2: Rigorous and Efficient Directed Greybox Fuzzing". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2022, pp. 2595–2609. DOI: 10.1145/3548606.3560648.

[266] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". *IEEE Access* (2017), pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

[267] Lwin Khin Shar, Lionel C. Briand, and Hee Beng Kuan Tan. "Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning". *IEEE Transactions on Dependable and Secure Computing* (2015), pp. 688–707. DOI: 10.1109/TDSC.2014.2373377.

[268] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". *IEEE Transactions on Software Engineering* (2011), pp. 772–787. DOI: 10.1109/TSE.2010.81.

[269] Yonghee Shin and Laurie Williams. "An Empirical Model To Predict Security Vulnerabilities Using Code Complexity Metrics". In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. ACM. 2008, pp. 315–317. DOI: 10.1145/1414004.1414065.

[270] Yonghee Shin and Laurie Williams. "Can Traditional Fault Prediction Models Be Used for Vulnerability Prediction?" *Empirical Software Engineering* (2013), pp. 25–59. DOI: 10.1007/s10664-011-9190-8.

[271] Perforce Software. *Klocwork: Best Static Code Analyzer for Developer Productivity, SAST, and DevOps/DevSecOps*. Mar. 1, 2024. URL: https://www.perforce.com/products/klocwork.

[272] Secure Software. *Rough Auditing Tool for Security (RATS)*. Mar. 1, 2024. URL: https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[273] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. "Rolecast: Finding Missing Security Checks When You Do Not Know What Checks Are". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM. 2011, pp. 1069–1084. DOI: 10.1145/2048066.2048146.

[274] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. "SoK: Sanitizing for Security". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[275] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. "A General Software Defect-Proneness Prediction Framework". *IEEE Transactions on Software Engineering* (2011), pp. 356–370. DOI: 10.1109/TSE.2010.90.

[276] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. "Inputs From Hell: Learning Input Distributions for Grammar-Based Test Generation". *IEEE Transactions on Software Engineering* (2022), pp. 1138–1153. DOI: 10.1109/TSE.2020.3013716.

[277] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. "One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-Tailored Program State Restriction". In: *Proceedings of the Annual Computer Security Applications Conference*. ACM. 2022, pp. 388–399. DOI: 10.1145/3564625.3564643.

[278] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. "An Empirical Study of Deep Learning Models for Vulnerability Detection". In: *Proceedings of the International Conference on Software Engineering*. IEEE. 2023, pp. 2237–2248. DOI: 10.1109/ICSE48619.2023.00188.

[279] Dominic Steinhöfel and Andreas Zeller. "Input Invariants". In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2022, pp. 583–594. DOI: 10.1145/3540250.3549139.

[280] W. Richard Stevens. *UNIX Network Programming*. Second. Prentice Hall, 1999.

[281] Maddie Stone. *0-Day In-the-Wild Exploitation in 2022... So Far*. June 29, 2022. URL: https://github.com/maddiestone/ConPresentations/blob/master/FIRST2022.2022_0days_so_far.pdf.

[282] Maddie Stone. *Déjà Vu-Lnerability: A Year in Review of 0-Days Exploited In-the-Wild in 2020*. Feb. 3, 2021. URL: https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html.

[283] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-Flow Analysis in LLVM". In: *Proceedings of the International Conference on Compiler Construction*. ACM. 2016, pp. 265–266. DOI: 10.1145/2892208.2892235.

[284] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. First. Pearson Education, 2007.

[285] Ari Takanen, Jared D. Demott, Charles Miller, and Atte Kettunen. *Fuzzing for Software Security Testing and Quality Assurance*. First. Artech House, 2008.

[286] AFL++ Team. *AFL++ Overview*. Jan. 1, 2024. URL: https://aflplus.plus/.

[287] AFL++ Team. *Fuzzing with AFL++: How long to fuzz a target?* Jan. 1, 2024. URL: https://aflplus.plus/docs/fuzzing_in_depth/.

[288] Ken Thompson. "Reflections on Trusting Trust". *Communications of the ACM* (8 1984), pp. 761–763. DOI: 10.1145/358198.358210.

[289] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. "To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools". In: *Proceedings of the International Conference on Automated Software Engineering*. ACM. 2012, pp. 50–59. DOI: 10.1145/2351676.2351685.

[290] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. "ALETHEIA: Improving the Usability of Static Security Analysis". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2014, pp. 762–774. DOI: 10.1145/2660267.2660339.

[291] Douglas A. Troy and Stuart H. Zweben. "Measuring the Quality of Structured Designs". *Journal of Systems and Software* (1981), pp. 113–120. DOI: 10.1016/0164-1212(81)90031-5.

[292] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. "Semi-Valid Input Coverage for Fuzz Testing". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 56–66. DOI: 10.1145/2483760.2483787.

[293] Stephen Turner. "Security Vulnerabilities of the Top Ten Programming Languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby". *Journal of Technology Research* (2014), pp. 1–16.

[294] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C. Gall. "Context Is King: The Developer Perspective on the Usage of Static Analysis Tools". In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. IEEE. 2018, pp. 38–49. DOI: 10.1109/SANER.2018.8330195.

[295] Andreas Wagner and Johannes Sametinger. "Using the Juliet Test Suite To Compare Static Security Scanners". In: *Proceedings of the International Conference on Security and Cryptography*. SCITEPRESS. 2014, pp. 244–252. DOI: 10.5220/0005032902440252.

[296] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". In: *Proceedings of the Network and Distributed System Security Symposium*. Internet Society. 2000, pp. 3–17.

[297] James Walden, Jeff Stuckman, and Riccardo Scandariato. "Predicting Vulnerable Components: Software Metrics vs Text Mining". In: *Proceedings of the International Symposium on Software Reliability Engineering*. IEEE. 2014, pp. 23–33. DOI: 10.1109/ISSRE.2014.32.

[298] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. "Typestate-Guided Fuzzer for Discovering Use-After-Free Vulnerabilities". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2020, pp. 999–1010. DOI: 10.1145/3377811.3380386.

[299] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing". In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. USENIX. 2019, pp. 1–15.

[300] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. "Software Testing With Large Language Models: Survey, Landscape, and Vision". *IEEE Transactions on Software Engineering* (2024), pp. 911–936. DOI: 10.1109/TSE.2024.3368208.

[301] Mingzhe Wang, Jie Liang, Chijin Zhou, Yuanliang Chen, Zhiyong Wu, and Yu Jiang. "Industrial Oriented Evaluation of Fuzzing Techniques". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE. 2021, pp. 306–317. DOI: 10.1109/ICST49551.2021.00043.

[302] Pengfei Wang, Xu Zhou, Tai Yue, Peihong Lin, Yingying Liu, and Kai Lu. "The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing". *Software Testing, Verification and Reliability* (2024). DOI: 10.1002/stvr.1869.

[303] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. "Deep Semantic Feature Learning for Software Defect Prediction". *IEEE Transactions on Software Engineering* (2020), pp. 1267–1293. DOI: 10.1109/TSE.2018.2877612.

[304] Song Wang, Taiyue Liu, and Lin Tan. "Automatically Learning Semantic Features for Defect Prediction". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2016, pp. 297–308. DOI: 10.1145/2884781.2884804.

[305] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". In: *Proceedings of the Network and Distributed System Security Symposium*. Internet Society. 2020. DOI: 10.14722/ndss.2020.24422.

[306] Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. "SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing". In: *Proceedings of the Asia Conference on Computer and Communications Security*. ACM. 2024, pp. 1539–1553. DOI: 10.1145/3634737.3661141.

[307] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. "Memlock: Memory Usage Guided Fuzzing". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2020, pp. 765–777. DOI: 10.1145/3377811.3380396.

[308] Elaine J. Weyuker. "Axiomatizing Software Test Data Adequacy". *IEEE Transactions on Software Engineering* (1986), pp. 1128–1138. DOI: 10.1109/TSE.1986.6313008.

[309] Elaine J. Weyuker and Thomas J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains". *IEEE Transactions on Software Engineering* (1980), pp. 236–246. DOI: 10.1109/TSE.1980.234485.

[310] David A. Wheeler. *Flawfinder*. Mar. 1, 2024. URL: https://dwheeler.com/flawfinder/.

[311] Tzu T. Wong and Po Y. Yeh. "Reliable Accuracy Estimates From k-Fold Cross Validation". *IEEE Transactions on Knowledge and Data Engineering* (2020), pp. 1586–1594. DOI: 10.1109/TKDE.2019.2912815.

[312] Valentin Wüstholz and Maria Christakis. "Targeted Greybox Fuzzing With Static Lookahead Analysis". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2020, pp. 789–800. DOI: 10.1145/3377811.3380388.

[313] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. "HYDRA: Massively Compositional Model for Cross-Project Defect Prediction". *IEEE Transactions on Software Engineering* (2016), pp. 977–998. DOI: 10.1109/TSE.2016.2543218.

[314] Yichen Xie, Andy Chou, and Dawson Engler. "ARCHER: Using Symbolic, Path-Sensitive Analysis To Detect Memory Access Errors". In: *Proceedings of the Symposium on the Foundations of Software Engineering*. ACM. 2003, pp. 327–336. DOI: 10.1145/940071.940115.

[315] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. *Large Language Models for Cyber Security: A Systematic Literature Review*. 2024. arXiv: 2405.04760 [cs.CR].

[316] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities With Code Property Graphs". In: *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 590–604. DOI: 10.1109/SP.2014.44.

[317] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning". In: *Proceedings of the USENIX Workshop on Offensive Technologies*. USENIX. 2011.

[318] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. "Generalized Vulnerability Extrapolation Using Abstract Syntax Trees". In: *Proceedings of the Annual Computer Security Applications Conference*. ACM. 2012, pp. 359–368. DOI: 10.1145/2420950.2421003.

[319] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. "Chucky: Exposing Missing Checks In Source Code for Vulnerability Discovery". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2013, pp. 499–510. DOI: 10.1145/2508859.2516665.

[320] Jinqiu Yang, Lin Tan, John Peyton, and Kristofer A. Duer. "Towards Better Utilizing Static Application Security Testing". In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. IEEE. 2019, pp. 51–60. DOI: 10.1109/ICSE-SEIP.2019.00014.

[321] Qian Yang, J. Jenny Li, and David M. Weiss. "A Survey of Coverage-Based Testing Tools". *The Computer Journal* (2007), pp. 589–597. DOI: 10.1093/comjnl/bxm021.

[322] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. "Deep Learning for Just-in-Time Defect Prediction". In: *Proceedings of the International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 17–26. DOI: 10.1109/QRS.2015.14.

[323] Wei You, Peiyuan Zong, Kai Chen, Xiao Feng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. "SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits". In: *Proceedings of the Conference on Computer and Communications Security*. ACM. 2017, pp. 2139–2154. DOI: 10.1145/3133956.3134085.

[324] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2020, pp. 2307–2324.

[325] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2018, pp. 745–761.

[326] Michał Zalewski. *American Fuzzy Lop (AFL)*. Jan. 1, 2024. URL: https://lcamtuf.coredump.cx/afl/.

[327] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. "Towards Building a Universal Defect Prediction Model". In: *Proceedings of the Working Conference on Mining Software Repositories*. ACM. 2014, pp. 182–191. DOI: 10.1145/2597073.2597078.

[328] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse Hsun Chen, and Ahmed E. Hassan. "A Study of C/C++ Code Weaknesses on Stack Overflow". *IEEE Transactions on Software Engineering* (2022), pp. 2359–2375. DOI: 10.1109/TSE.2021.3058985.

[329] Jia Ming Zhang, Zhan Qi Cui, Xiang Chen, Huan Huan Wu, Li Wei Zheng, and Jian Bin Liu. "DeltaFuzz: Historical Version Information Guided Fuzz Testing". *Journal of Computer Science and Technology* (2022), pp. 29–49. DOI: 10.1007/s11390-021-1663-7.

[330] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. "Boosting Spectrum-Based Fault Localization Using Pagerank". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM. 2017, pp. 261–272. DOI: 10.1145/3092703.3092731.

[331] Yucheng Zhang and Ali Mesbah. "Assertions Are Strongly Correlated With Test Suite Effectiveness". In: *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM. 2015, pp. 214–224. DOI: 10.1145/2786805.2786858.

[332] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. "Combining Software Metrics and Text Features for Vulnerable File Prediction". In: *Proceedings of the International Conference on Engineering of Complex Computer Systems*. IEEE. 2015, pp. 40–49. DOI: 10.1109/ICECCS.2015.15.

[333] Yuyue Zhao, Yangyang Li, Tengfei Yang, and Haiyong Xie. "Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning". In: *Proceedings of the International Conference on Information Security and Cryptology*. Springer. 2020, pp. 134–153. DOI: 10.1007/978-3-030-42921-8_8.

[334] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. "FishFuzz: Catch Deeper Bugs by Throwing Larger Nets". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2023, pp. 1343–1360.

[335] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. "On the Value of Static Analysis for Fault Detection in Software". *IEEE Transactions on Software Engineering* (2006), pp. 240–253. DOI: 10.1109/TSE.2006.38.

[336] Xiaogang Zhu and Marcel Böhme. "Regression Greybox Fuzzing". In: *Proceedings of the Conference on Computer and Communications Security*. 2021, pp. 2169–2182. DOI: 10.1145/3460120.3484596.

[337] Thomas Zimmermann and Nachiappan Nagappan. "Predicting Defects Using Network Analysis on Dependency Graphs". In: *Proceedings of the International Conference on Software Engineering*. ACM. 2008, pp. 531–540. DOI: 10.1145/1368088.1368161.

[338] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. "Cross-Project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process". In: *Proceedings of the Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. 2009, pp. 91–100. DOI: 10.1145/1595696.1595713.

[339] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. IEEE. 2010, pp. 421–428. DOI: 10.1109/ICST.2010.32.

[340] Misha Zitser, Richard Lippmann, and Tim Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code". In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. ACM. 2004, pp. 97–106. DOI: 10.1145/1029894.1029911.

[341] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. "FuzzGuard: Filtering Out Unreachable Inputs in Directed Grey-Box Fuzzing Through Deep Learning". In: *Proceedings of the USENIX Security Symposium*. USENIX. 2020, pp. 2255–2269.

List of Acronyms

PSO particle swarm optimization 71
RAM random-access memory 41
RcMO root cause-manifestation overlap 37
RDF random decision forest 54
ROC receiver operating characteristic 17
SARIF Static Analysis Results Interchange Format 12
SAST static application security testing 6
SLES SUSE Linux Enterprise Server 72
SMT satisfiability modulo theories 71
SUT system under test 3
SVM support vector machine 55
TBB target basic block 93
TPR true positive rate 14
TTE time-to-error 109
VCS version control system 15

List of Figures

List of Tables

List of Listings

List of Algorithms