Hybrid Testing: Combining Static Analysis and Directed Fuzzing

by

Peyton Shields

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2023

Authored by: Peyton Shields
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Nathan Burow
Lincoln Laboratory Technical Staff, Thesis Supervisor

Certified by: Hamed Okhravi
Lincoln Laboratory Senior Staff, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Hybrid Testing: Combining Static Analysis and Directed Fuzzing

by

Peyton Shields

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

New CVEs are discovered each year and their underlying bugs leave applications vulnerable to exploitation. Software is still frequently written in bug prone languages, e.g. C and C++, and a single missed check during manual testing can result in vulnerabilities. Existing automated testing tools such as fuzzing are limited in scope or in the case of static analysis, have a high false positive rate. Without improved automated testing, it can be challenging for developers to debug large, complex codebases. In this paper, Hybrid Testing is presented as a solution. Hybrid Testing combines static and dynamic analyses, leveraging static analysis to perform complex reasoning about logic, memory management, and concurrency. It creates a novel orchestration system which allows us to automatically verify the output of static analysis tools using directed fuzzing. Hybrid Testing is the first vulnerability detection technique with full codebase coverage and no false positives. It can be seamlessly integrated into the development cycle and scales well to large codebases. This work details the design and implementation of Hybrid Testing and evaluates its performance across a corpus of open-source C and C++ applications in the Magma benchmark. Hybrid Testing aims to promote more secure software through rigorous testing, making it easier for developers to detect security issues. We demonstrate Hybrid Testing can find vulnerabilities up to 25% faster with 17% higher accuracy (when detecting additional bugs) than current automated testing strategies.

Thesis Supervisor: Nathan Burow
Title: Lincoln Laboratory Technical Staff

Thesis Supervisor: Hamed Okhravi
Title: Lincoln Laboratory Senior Staff

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Acknowledgments

I am grateful to Lincoln Lab's Secure Resilient Systems & Technology Group for the opportunity to pursue my Masters and conduct research at MIT. I would like to express my deepest gratitude to my thesis supervisors, Nathan Burow and Hamed Okhravi, for their continuous mentorship and invaluable feedback during the course of this work. I also wish to thank Derrick McKee for his insight into the implementation and design of Hybrid Testing. Thanks should also go to Christina Chase, who helped me navigate academic life during the entirety of my time at MIT as an undergraduate and graduate student.

I cannot begin to express my thanks to my parents, family, and friends for their emotional and moral support throughout the course of my studies at MIT. Their support and kindness helped me stay motivated and persistent, no matter the task ahead. I truly appreciate everyone who showed their support during the course of my time at MIT and am tremendously grateful for their impact on my life and career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

New vulnerabilities are discovered each year in critical applications. For example, CVE-2022-2602 [18] in the Linux Kernel enables local privilege escalation and CVE-2022-3602 [19] in OpenSSL can cause a denial of service. Erroneous, insecure software can have consequences, especially in a government or industry context, leaving applications vulnerable to exploitation. Data breaches due to insecure software are commonplace and could be mitigated through improved automatic testing. In 2015 alone, the U.S. Office of Personnel Management suffered multiple security breaches leaking sensitive information of approximately 25 million people [58]. There remains a need to improve software quality and assurance through rigorous testing.

A subset of current software testing practices rely on developers to manually create test cases to validate the behavior of a program. Two common testing practices are unit testing small, isolated code segments and integration testing the behavior of an entire application. Unit testing has the benefit of being fine-grained as it is targeted towards smaller code segments, but it is time-intensive to create as the number of lines of code increases. Integration testing, conversely, has higher code coverage than unit testing but it is coarse-grained as testing is performed at the application level [11]. All forms of manual test creation require developers to reason about possible inputs and edge cases which may be less obvious. As codebases grow larger and more complex, security issues and bugs slip through the cracks on a regular basis, often resulting in the aforementioned vulnerabilities. Testing and verifying lines of code can easily be overlooked and the complexity of a codebase brings new challenges during the testing process. A single missed edge case could result in a new vulnerability.

Automated testing is a promising solution to this problem as it can be seamlessly integrated into the development cycle, scales well to large codebases, and is capable of complex reasoning about logic, memory management, and concurrency. Automated testing typically requires minimal deployment effort from the developer compared to manual test case creation and acts as a secondary assurance metric during the development cycle. Within the growing repertoire of automated software testing tools, there are two predominant tool classes: static and dynamic.

Static analysis inspects an application's source code and attempts to flag bugs in the implementation without executing the application [17, 6]. Static analysis has been successfully deployed to large codebases such as Google and Facebook where it has helped prevent null-pointer dereferences, buffer overflows, and unsatifisable control logic [5, 23]. A core strength of static analysis is high code coverage. Static analyzers can examine an entire codebase, processing millions of lines of code at a time [9]. This efficiency is enabled in part because static analysis does not depend on execution traces, making it highly scalable. Static analyzers are capable of complex reasoning about logic, memory management, and concurrency. With this complexity comes the core weakness of static analysis: false positives. Without execution, it is often not possible to guarantee that a bug exists. Additionally, false positives diminish the effect of the analysis on the developer, who must decide to fix an issue based on the analyzer output [63, 9, 42]. There are numerous open-source and proprietary static analyzers which have been heavily evaluated and pitted against each other [22, 69]. Across each tool, one tenant is clear: there is no single best option.

Dynamic analysis attempts to find concrete inputs triggering a bug and is dependent on execution traces. In contrast to static analysis, dynamic analysis yields no false positives. A common sub-class of dynamic analysis, and the focus of this work, is fuzzing. Common fuzzers such as AFL [1], AFL++ [28], and libFuzzer [49] attempt to generate invalid or erroneous inputs resulting in a crash. A byproduct of a zero false positive rate is the time intensive nature of fuzzing. A program must be executed each time a new input is generated. Fuzzers operate with limited insight into the logic and control flow of the program. Canonically, inputs are randomly generated based on extracted control flow graphs and flow analysis. This implies a key limitation: fuzzers are only capable of testing code reachable by execution of a given input [84]. Fuzzing has also been widely deployed for open-source projects such as OpenSSL, Nginx, and QEMU which are part of Google's OSS-Fuzz, a community effort to deploy continuous dynamic analysis to open-source applications [33].

The two classes, static and dynamic, complement each other, the weaknesses of static analysis are the strengths of dynamic testing and vice versa. An overview of the trade-offs present in each class are depicted in Figure 1-1. Static analysis does not depend on execution traces and thus an entire codebase can be ingested and analyzed in a finite amount of time. Common static analyzers such as CodeChecker [27] and FlawFinder [29] can detect bugs like unsatisfiable logic and infinite loops, null pointer dereferences and use-after-free bugs, as well as data races due to concurrency. The downside of such ubiquity and generality is a large number of results as shown in Table 5.1, 20% [78] or more of which could be false positives. Dynamic analysis is the counterpart to static analysis in this sense because it has a zero false positive rate. Dynamic analysis, specifically fuzzing, observes execution traces and attempts to generate inputs causing a program to crash. These traces serve as feedback to inform future input generation, enabling dynamic analysis to explore different paths of execution. Additionally, by using concrete input values, dynamic analysis explores data and control flows which occur in practice. The downside of dynamic analysis is an unbounded amount of time to detect a bug. The code may be bug free or the fuzzer may be incapable of reaching the problematic line of code from the program's entry-point.

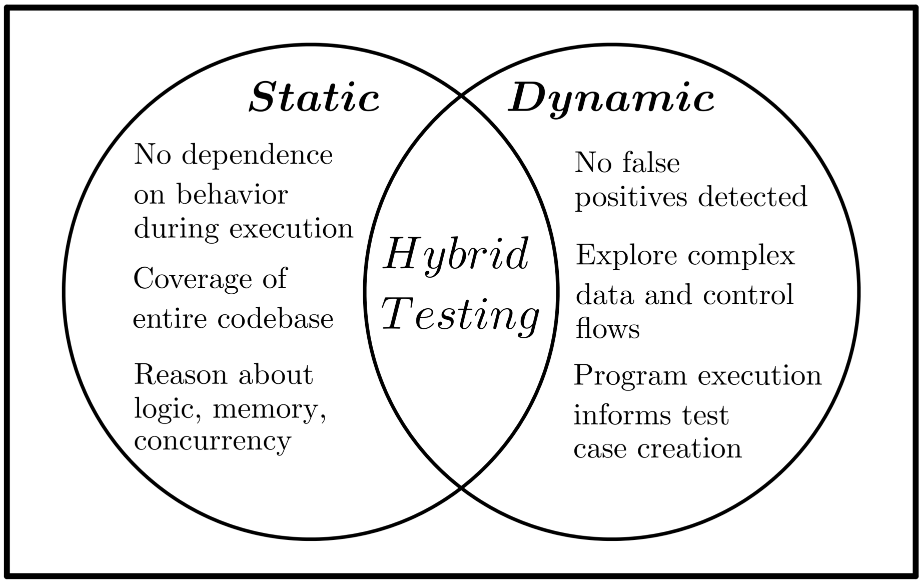
Static
No dependence on behavior during execution
Coverage of entire codebase
Reason about logic, memory, concurrency
Hybrid Testing
Dynamic
No false positives detected
Explore complex data and control flows
Program execution informs test case creation

Figure 1-1: Comparison of Static vs. Dynamic Testing

One of the pitfalls of static analysis is the high false positive rate [54], often 20% [78] or more of all static analysis results. This can be reduced or eliminated entirely with the advent of a new type of dynamic analysis: directed fuzzing [10, 65, 57, 16, 40, 85, 45, 25, 35, 48, 70]. Prior to this work, the findings of a static analyzer typically had minimal bearing on the results of a dynamic analysis tool [84]. Specifically, fuzzing tools were not informed by the errors detected during static analysis. The recent innovation of directed fuzzing enables their combination, forming a synergy between the strengths of one and the weaknesses of the other. Combining static analysis with directed fuzzing allows fuzzing target lines of code which are more likely to contain a bug (according to static analysis) and thus more likely to cause a crash.

Directed fuzzers function much like their undirected counterparts, but can probe specific code segments by attempting to generate inputs which drive execution towards a problematic line of code. For example, AFLGo [10] calculates the distance to a target location and uses simulated annealing to craft inputs which come closer to the target. SieveFuzz [70] contrastingly prunes infeasible paths at runtime via statically generated preconditions to reach a target.

To date, research in automated testing has largely remained siloed within each class (static and dynamic analysis). Combining analyses is a promising step towards reducing false positives. At the time of this writing, we are only aware of two works in this space: Arbiter [74] and Batg [75]. Arbiter and Batg attempt to bridge static and dynamic analysis, but neither leverage fuzzing. Both techniques leverage static analysis to detect vulnerabilities and use symbolic execution (see Section 2.3) to find triggering inputs for said vulnerabilities.

With the growing research in directed fuzzing, the time is ripe to combine static analysis and directed fuzzing. The high code coverage of static analysis and the zero false positive rate of directed fuzzing are a complementary match. A static analyzer can first examine a codebase, highlighting areas likely to contain security issues, and use this to inform the directed fuzzer. The directed fuzzer can then better reason about the program's behavior, crafting inputs to reach the target areas specified by the static analyzer.

This work utilizes static analyzers to examine an application for likely bugs, extract the relevant source code line numbers and filenames, and provide this information to a directed fuzzer. The directed fuzzer then fuzzes the application, attempting to trigger the bugs detected by the static analyzer and produce a crash. An orchestration layer parses static analyzer output and generates an intermediate representation we call the Static Analysis Intermediate Representation (SA-IR). The orchestration layer prioritizes the SA-IR for fuzzing based on the static analyzer's predicted likelihood a vulnerability exists.

We employ exemplar static analysis tools CodeChecker [27] and FlawFinder [29] to mitigate the possibility that a single static analyzer does not detect a likely security issue. We use directed fuzzers AFLGo [10] and SieveFuzz [70] to evaluate the existence of a bug, categorizing it as a true or false positive. Multiple directed fuzzers are selected to avoid the same bias towards a single tool. To conduct this research, we use C and C++ applications from the Magma fuzzing benchmark [36] to provide ground truth data for both the static analyzers and directed fuzzers.

This work creates a new class of software testing techniques with full code coverage and zero false positives by integrating static analysis with directed fuzzing. We name this technique Hybrid Testing and demonstrate that it can automatically detect and triage bugs. We evaluate Hybrid Testing's ability to reduce static analyzer false positives across a corpus of open-source applications in the Magma benchmark and compare its detection capabilities to AFL [1]. We show that Hybrid Testing finds 17% more bugs than AFL and can detect bugs up to 25% faster. We demonstrate that Hybrid Testing is a step towards finding bugs with zero false positives. We aim for Hybrid Testing to improve software quality and security by expanding the code coverage of automated testing and eliminating false positives.

Chapter 2

Related Work

Recent research in automated testing has primarily focused on a single class of tools. This work broadly categorizes these classes as static analysis, dynamic analysis in the context of fuzzing, and symbolic execution. This chapter aims to provide an overview of research in an individual tool class as well as work on combining tool classes to improve automated testing. To date, there has been little related work on the focus of this thesis: combining directed fuzzing and static analysis to eliminate false positives with full codebase coverage.

2.1 Static Analysis

Static analysis, in the context of this work, examines a program's source code and attempts to detect errors in the implementation by pattern matching, solving constraints, or attempting to prove theorems regarding concurrency, data flow, or general program state [17, 6]. Static analysis has no dependence on execution traces and can detect common weaknesses described by the MITRE CWE [2, 52]. Previous work has shown that static analysis can assist in early detection of software bugs, but existing tools are imperfect, often yielding many reports with a high degree of false positives [83, 55, 76].

Current efforts aiding in the evaluation of static analysis tools and their respective accuracies include NIST SAMATE [34], NIST SARD [64], NIST SATE [22], and FAULTBENCH [37]. Software Assurance Metrics and Tool Evaluation (SAMATE) catalogs static analyzers and classifies bugs detectable through such tools. The Software Assurance Reference Dataset (SARD) is a ground truth static analysis benchmark which can be used to measure the accuracy of a tool. The Static Analysis Tool Exposition (SATE) presents current static analysis tools, how they are deployed in the real world, and their capabilities and limitations. FAULTBENCH is an additional static analysis benchmark geared towards precisely specifying false positives and evaluating false positive mitigation techniques.

The effective integration of static analysis into the development workflow has been a topic of significant research. Large codebases prove especially tricky as static analyzers frequently report many potential errors, false or true positive status unknown [13]. If a developer is unable to understand or easily digest these errors, it was found that they would often go unfixed [63, 23, 72]. At Google and Facebook, static analysis was integrated into the code review process, displaying only the most likely of errors side-by-side with code reviews. This improved bug tractability and helped detect many common bugs such as buffer overflows and null pointer dereferences.

Significant work on static analysis tools themselves has focused on usability, namely making errors understandable and reducing false positives [9, 42]. Though static analyzers are capable of complex analyses, it is ultimately the developer who must validate the bug because it could be a false positive. The interfaces presented by static analyzers as well as their easy integration into the development cycle were found to be top priorities for usability, matching the observations at Facebook and Google [68].

To better understand existing false positive patterns, previous evaluations of static analysis also identify and categorize false positives by common code structures including memory leaks originating from an unsatisfiable condition, file descriptor management, and reading from or inserting into a buffer [62]. Research on false positive detection and classification found trends in the false positive rate based on CWE, source code origin, and location within a program [13]. A recurring theme across recent work is identifying potential false positives before attempting to mitigate a tool warning.

Combating the high degree of false positives has been a key area of work in promoting more secure software. Example strategies include interactively ranking the likelihood a warning is a false positive, using model checking to generate assertions for each warning, and classifying false positives through machine learning via neural networks [53, 46, 72]. Despite existing elimination strategies, no method is full-proof. There still exists a need for full codebase coverage with no false positives.

Work on the analytical capabilities of static analyzers has continued since their inception. The Static Analysis Symposium (SAS) showcases recent such advancements which include optimizing for performance, making the analysis interactive rather than generating reports to later be evaluated, and improvements to concurrency analysis [39, 66]. Tools such as Calysto [8], an interprocedurally path sensitive analyzer, and TRACER [43], a tool performing signature based taint analysis of recurring code segments, are being developed and deployed to the wider software testing ecosystem. Despite advancements in analytical capabilities, developers frequently abandon tools due to false positives and cumbersome workflows.

2.2 Fuzzing-Based Dynamic Analysis

In contrast to static analysis, fuzzing is highly dependent on execution traces. Fuzzing attempts to force a program into an erroneous state, typically leading to a crash, by executing a program with generated, malformed inputs [51]. These erroneous inputs are often generated based on the initial input, or seeds, fed to the fuzzer. Mutation-based fuzzing is a popular technique which applies transformations to the initial seeds. Recent work has shown that the seeds selected for fuzzing can heavily impact the number of bugs detected [38]. For example, MOPT [50] is a mutation scheduling algorithm for selecting seeds which has demonstrated improved performance in comparison to AFL [1]. SLF [80], on the other hand, attempts to create valid inputs when minimal initial seed data is available.

Fuzzing techniques can be further broken down into three main families: black-box, grey-box, and white-box [26, 31, 14, 47, 51]. Black-box fuzzing requires no knowledge of a program's source code and relies solely on knowledge of crashes to generate inputs and detect bugs. Without knowledge of a program's internals, the fuzzer is unable to determine if an input expanded code coverage. This implies that a large selection of input seed data may be necessary for black-box fuzzing to be effective. Example use cases of black-box fuzzing include testing applications with complex input formats such as SQL or the HTML DOM [3]. Web applications are predominant subjects for black-box fuzzing.

White-box fuzzing requires full access to source code and often uses in-depth program analysis to reason about a program's behavior and generate corresponding inputs. The overhead required for white-box fuzzing is typically much higher than black-box fuzzing due to this dependence on program internals [51]. White-box fuzzing functions similarly to concolic execution (see Section 2.3) in that it records the branch constraints along the execution trace for a given input. It then attempts to create a new input by negating and mutating these constraints [47]. SAGE [32] is one such popular white-box fuzzer.

Grey-box fuzzing lies in the middle and is the primary focus of this work. Popular grey-box fuzzers include AFL [1], AFL++ [28], and libFuzzer [49]. Grey-box fuzzers require some knowledge of program internals and use code instrumentation to detect the code coverage of a generated input during execution. Based on the instrumentation output, fuzzers commonly use mutation strategies including bit flips, copying, and deleting to create new inputs [47, 51]. Recent work on grey-box fuzzing include GREYONE [30], MUZZ [15], CONZZER [41], and SGFuzz [7]. GREYONE is a data flow sensitive fuzzer which applies taint analysis to reach code regions typically unexplored under normal execution conditions. MUZZ and CONZZER, on the other hand, are designed to fuzz concurrent applications by exploring different thread interleavings and attempting to detect data races or leaks. SGFuzz aims to explore bugs resulting from complex program states, or bugs only occuring after a specific sequence of inputs are fed to a program.

A common thread among many grey-box fuzzing techniques is coverage guided fuzzing (CGF) which uses program instrumentation to detect code segments fuzzing has explored and attempts to increase code coverage. Recent work on techniques for CGF include AFLsmart [60] and MobFuzz [82]. AFLsmart improves upon existing mutation strategies for test case generation through input structure awareness, providing a higher level of reasoning about the initial seeds. MobFuzz takes a new approach by modeling fuzzing as a multi-objective problem instead of the single objective to increase code coverage. This is accomplished by adaptively selecting which objective to optimize for given a generated input, including speed, input size, and potential reward for a given input. In contrast to AFLsmart and MobFuzz, UnTracer [56] and *Fine-Grained Coverage-Based Fuzzing* [79] offer improvements to existing coverage analysis techniques. UnTracer reduces the overhead incurred during fuzzing with additional instrumentation on how much of the codebase has currently been explored during a fuzzing campaign. *Fine-Grained Coverage-Based Fuzzing* differentiates itself by examining control and data flow rather than overall branch coverage to determine if a fuzzer test case increased code coverage.

The advent of directed grey-box fuzzing has helped to deliver new work on code coverage. Directed fuzzing can be further broken down by the techniques used to drive execution towards a specific code segment. A common technique across all fuzzers cited is a reachability analysis. This is generally performed by extracting a control flow graph and computing the distance from a program's entry point to the target location's basic block. AFLGo [10], UAFuzz [57], Hawkeye [16], Windranger [25], and LOLLY [48] implement novel seed selection algorithms based on coverage information at runtime. Beacon [40] and SieveFuzz [70] make use of static analysis to prune impossible paths, greatly reducing the input space. MC2 [65] is a provably efficient directed fuzzer which relies heavily on Monte Carlo simulation and a randomized binary search algorithm. CAFL [45] and Dowser [35] combine symbolic execution to constrain and create new inputs. Finally, FuzzGuard [85] approaches fuzzing from an AI perspective and uses deep learning to compute likely inputs.

While directed fuzzing in the context of this work attempts reach a specific code region, other forms of guided fuzzing include ParmeSan [59] and MemLock [77]. ParmeSan uses santizers such as AddressSantizer (ASan) and UndefinedBehaviorSanitizer (UBSan) to guide fuzzing towards code segments likely to violate sanitizer checks. MemLock, contrastingly, is designed to generate test cases with high memory consumption and cause corruption.

Fuzzers are commonly paired with sanitizers such as address sanitization (ASan) to amplify detection of different classes of bugs. Not all bugs lead to program crashes; pairing fuzzing with a sanitizer can help detect bugs which may otherwise slip through the cracks. Even if a program does crash, the bug could have occurred much earlier in runtime which further complicates root cause analysis. Sanitizers often act as an early or additional detection mechanism during the fuzzing process.

2.3 Symbolic Execution

Symbolic execution is a mathematical testing technique which reasons about inputs at a given code segment on all possible execution paths by applying constraints to the input. It then uses a constraint or satisfiable modulo theory (SMT) solver such as Z3 [81] to solve for possible inputs. The downside of symbolic execution is that it must track all the constraints for a given input, which can quickly explode as the execution path expands. At this point, symbolic execution cannot reason deeper into the code as the constraint solver is unable to find a solution. To combat this, many symbolic execution tools perform concolic execution, augmenting their analysis with concrete inputs to relax constraints. Popular symbolic execution tools include Angr [67] and KLEE [12] which are capable of more complex reasoning than fuzzing, at the cost of constraint explosion. New tools such as SymCC [61], which embeds concolic execution instrumentation directly into a compiled binary, are also being developed within the symbolic execution testing landscape.

2.4 Combining Analyses

Hybrid Testing is not alone in its goal to create a scalable automated testing tool with no false positives. Arbiter [74] is a black-box binary analysis tool which combines static analysis and symbolic execution. Arbiter has proven capable of detecting CWEs such as unchecked return values (CWE-252), uncontrolled format strings (CWE-134), and predictable seeds in a PRNG (CWE-337). It operates by statically analyzing a binary for predefined conditions such as the CWEs above and uses symbolic execution to derive an input to reach the code segments from static analysis.

Batg [75] is another automated testing tool which combines static analysis and symbolic execution. Batg detects vulnerabilities by creating a test suite to bound the behavior of a program. Unlike Arbiter, Batg is a white-box testing tool. It requires source code for static analysis to detect potential program errors then uses symbolic execution to explore all inputs along the path to those errors.

Chapter 3

Design

Hybrid Testing combines static and dynamic analysis by using directed fuzzing to fuzz the target lines of code flagged during static analysis. It aims to automatically verify true positive static analysis targets and eliminate false positives. Our goals are for Hybrid Testing to provide full codebase coverage with complex reasoning about potential bugs, a core strength of static analysis, with the zero false positive rate of fuzzing. We aim for Hybrid Testing to promote more secure software by detecting deeper bugs than current testing techniques.

Hybrid Testing must be fully modular, supporting interchangeable static analyzers and directed fuzzers. This enables us to generalize Hybrid Testing to any toolkit (any subset of static analyzers or directed fuzzers used for automated testing). A new static analyzer or directed fuzzer should be easily integratable into the workflow. Hybrid Testing must also be fully automated. To support this, it should be capable of ingesting the output from static analysis and using this to begin directed fuzzing a given target code location. Modularity and automation is a core requirement for adoption of new testing techniques.

Hybrid Testing requires three component pieces: a set of static analyzers, an orchestration layer, and a set of directed fuzzers. The components are connected sequentially as shown in Figure 3-1, allowing each stage to be fully modular. Figure 3-1 shows that Hybrid Testing supports interchangeable static analyzers and directed fuzzers. Our pipeline runs a static analyzer then parses and ingests its output using an orchestration layer. The orchestration layer acts as an intermediary between static analysis and a directed fuzzer, translating the outputs of one to the input of the other. The orchestration layer is responsible for generating a novel Static Analysis Intermediate Representation (SA-IR) and lowering the SA-IR

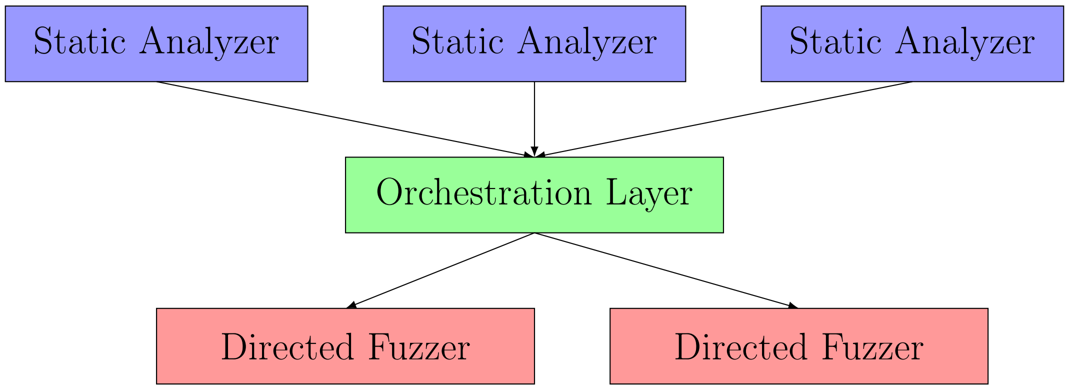

Figure 3-1: Hybrid Testing Pipeline Design

to any directed fuzzer. In this chapter, we examine the properties each component must have to automate the Hybrid Testing workflow. We also introduce the novel Static Analysis Intermediate Representation (SA-IR) and explain the choices we made to enable lowering the SA-IR to any directed fuzzer.

3.1 Static Analyzer Requisite Properties

The first component in the Hybrid Testing toolchain is static analysis. Figure 3-1 shows how the results of static analysis are then fed to the orchestration layer. We require a static analyzer to maintain the following properties for compatibility with Hybrid Testing:

- Provide full codebase coverage and be executable from the command line.
- Detect weaknesses which can be validated using existing santizers.
- Assign a severity to detected results and output results in a standardized format.

We require a static analyzer to provide full codebase coverage to identify potential vulnerabilities anywhere within a program's source code. We assume that a static analyzer outputs vulnerabilities which then act as input for directed fuzzing. Full codebase coverage also allows Hybrid Testing to meet its goal of zero false positives because Hybrid Testing can exhaustively fuzz all static analysis targets.

We further require the vulnerabilities found during static analysis to trigger an existing sanitizer. This allows us to verify if a bug actually exists, or if the static analysis result is a false positive. If a bug exists, an existing sanitizer such as address sanitization should detect the bug. A key advantage of this approach is that Hybrid Testing only has to run a sanitizer when a directed fuzzer reaches a bug location reported by static analysis. Sanitizers are time intensive and computationally expensive, allowing us to prioritize fuzzing additional static analysis targets instead of running every fuzzing input with sanitization enabled.

Assigning a severity to static analysis targets and standardizing output allows Hybrid Testing to fully automate bug detection. We can prioritize which static analysis targets to fuzz based on severity and easily integrate new tools into Hybrid Testing. This allows Hybrid Testing to generalize to any static analyzer. While it is not strictly necessary for a tool to be executable from the command line, it enables the creation of a fully automated pipeline without manually loading and storing results.

3.2 Orchestration Layer and SA-IR Overview

The second component in the Hybrid Testing toolchain is the orchestration layer as shown in Figure 3-1. The orchestration layer is responsible for parsing the output of a static analyzer then using those results to inform directed fuzzing. The orchestration layer must be modular, allowing for easy integration of a new parser for each static analyzer's results. It should accept configuration parameters on which static analyzers and directed fuzzers will be run. This allows the orchestration layer to select a parser and select how to lower static analysis results.

Hybrid Testing's orchestration layer design is centered around the SA-IR. The SA-IR is a lightweight, simplified format used to extract metadata from any static analyzer and is capable of being lowered to a directed fuzzer's input format. To meet these requirements, the SA-IR must minimally have the following properties:

- Uniquely identify the location of a static analysis result within a codebase — including file name, line number, and relative path within the codebase.
- Identify which static analyzer produced the SA-IR to refine results by tool.
- Categorize the SA-IR by a bug class, such as by CWE.
- Identify the severity of static analyzer warning, allowing prioritization of flagged results by severity thresholds.

We require the SA-IR to uniquely identify a static analysis target so it can be lowered to any directed fuzzer. A directed fuzzer must know exactly which code region it should attempt to fuzz and the SA-IR retains the full metadata associated with a given line of code. Regardless of the input format required by a directed fuzzer, the SA-IR can pinpoint an exact line of code and the orchestration layer can transform this target line into the required format. It is also worth noting that a SA report may be in the vicinity of a bug, but not at the exact line of code where the bug is detected. This is discussed further in Section 4.4 and may necessitate a root cause analysis for why a bug actually occurred. We discuss the root cause analysis we performed in Section 5.5.

Categorizing the SA-IR by bug class, severity, and analyzer are essential for meta-analysis and prioritization. We can prioritize which static analysis results to fuzz by examining the severity stored within the SA-IR. For example, Hybrid Testing could target only static analysis results deemed high severity. We can also prioritize by which bug class was assigned to the SA-IR; this provides additional flexibility by allowing us to fuzz only memory or concurrency issues, for example. Tagging the SA-IR with the analyzer that produced the report allows for report generation and gathering statistics on which tool produced the SA-IR. The fields in the SA-IR ensure that the orchestration layer is highly configurable and customizable to different toolkits.

3.3 Directed Fuzzing Requirements

The final component in the Hybrid Testing toolchain is directed fuzzing (see Figure 3-1). We require a directed fuzzer to be executable from the command line and to accept input on which code segment should be fuzzed. The fuzzer should be able to instrument a variety of open-source applications, accept a standardized input format, and output relevant data related to crashes. It should also be noted that Hybrid Testing is sensitive to the performance of a directed fuzzer. To apply Hybrid Testing, a directed fuzzer must be able to run an application and attempt to reach a static analysis target. Hybrid Testing's performance could vary significantly due to the underlying capabilities of a directed fuzzer to run a program and reach a target. Ideally, a directed fuzzer would also produce data flows and statistics if a target line of code was reached.

Hybrid Testing aims to mimic current fuzzing techniques in that it is fully automatic and should require minimal user intervention. Due to the flexibility of the SA-IR, Hybrid Testing imposes looser requirements on which directed fuzzers are compatible. This enables easier integration of fuzzers and preserves the modular design of Hybrid Testing.

Chapter 4

Implementation

Hybrid Testing combines static analysis with directed fuzzing to automatically analyze a codebase to detect bugs and begin fuzzing a region of code based on the static analysis. The Hybrid Testing pipeline is fully modular, supporting interchangeable static analyzers and directed fuzzers. This work implements a novel Static Analysis Intermediate Representation (SA-IR) which enables translation from a static analyzer and can then be lowered to the format required for directed fuzzing. Hybrid Testing is highly generalizable to different tool kits due to this modular setup.

Hybrid Testing is implemented as a three stage pipeline. A pipeline diagram is shown in Figure 4-1. The first stage is static analysis of a codebase, the second is an orchestration layer responsible for generating the SA-IR and lowering it to the format required for directed fuzzing, and the final stage instruments and fuzzes a program. Each pipeline stage is separated to allow easy integration of new tools as well as inspection of static analysis results, the SA-IR, and fuzzing results. The role of each component is explained in the relevant section. To evaluate Hybrid Testing, the ground truth fuzzing benchmark Magma [36] is used. This work makes multiple enhancements to Magma to support Hybrid Testing, including integrating directed fuzzing and static analysis.

4.1 Static Analysis

The first step in detecting bugs with Hybrid Testing runs two open-source static analyzers, CodeChecker [27] and FlawFinder [29], on a codebase. These static analyzers were selected for their widespread usage and documentation, compatibility with C and C++ code, and

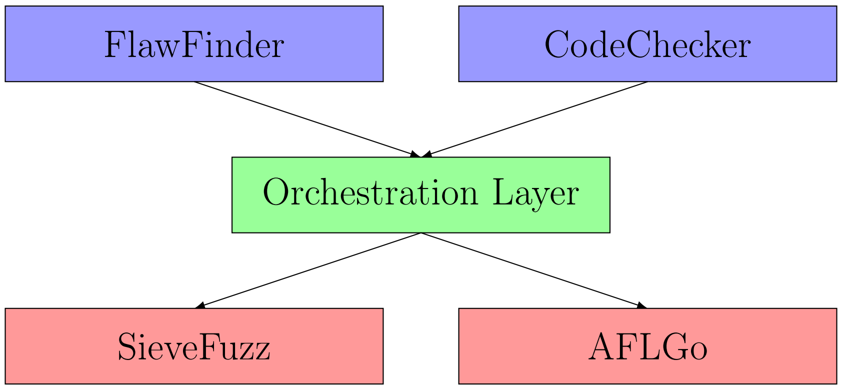

Figure 4-1: Hybrid Testing Pipeline Implementation

easy integration into the development workflow from the command line. However, these analyzers are only a subset of possible tools due to the modular design. Throughout the development process, we observed that CodeChecker and FlawFinder often found different bugs. This is consistent with prior evaluations of static analysis tools [22, 4]. To minimize the impact of a single tool's shortcomings on Hybrid Testing, this work combines two unique tools running in parallel to increase the probability of detecting a bug and reduce false negatives. Expanding the number of static analyzers used changes false positive mitigation strategies, expanding the design space of static analysis tools and potentially enabling better bug detection with Hybrid Testing.

CodeChecker is built on top of the LLVM Clang Analyzer and requires building an application before static analysis. This has the benefit of inspecting compiler commands and generating a compilation database. FlawFinder, contrastingly, is a Python framework which does not require compilation. The combination of the two provide Hybrid Testing with unique insights into the source code and the compiled application. Combining multiple tools promotes generalizability and in practice yielded more potential bugs than a single static analyzer. Hybrid Testing is only able to validate vulnerabilities detected through static analysis and more potential bugs allows for more rigorous testing of an application.

Hybrid Testing integrates two exemplar static analyzers for the purpose of this work; further evaluation of the ability of a static analyzer to detect a bug is out of scope. FlawFinder and CodeChecker support detection of many common weaknesses such as use-after-free, buffer overflows, and null pointer dereferences [29, 73]. By capturing a wide class of weaknesses through the combination of two static analyzers, this work expands the design space

Table 4.1: FlawFinder Output Format

File	Line	Column	Level	Category	Name
libxml2/repo/os400/dlfcn/dlfcn.c	466	22	5	race	readlink

of Hybrid Testing, augmenting the number of code regions to target.

A shortcoming of multiple static analysis tools is the differing output formats across tools. Each analyzer is unique and may provide more or less metadata when reporting a potential bug. Figure 4-2 shows a sample output from CodeChecker in JSON format. CodeChecker reports bug locations, a message about the potential vulnerability, an estimated severity, as well as events along the path to the bug. Table 4.1 shows sample output from FlawFinder in CSV format. Compared to CodeChecker, FlawFinder outputs similar metadata with the exception of bug path events. The varying output formats require writing a new parser for each static analyzer integrated into Hybrid Testing. The parsers for CodeChecker and FlawFinder require 50 to 100 lines of Python code and use simple tokenization to extract the metadata necessary for the SA-IR. Writing a new parser requires minimal development, provided that a static analyzer produces a standardized output format which can be reliably parsed. The orchestration layer implements such parsers to support translation from static analysis into SA-IR. Figure 4-1 details the static analysis stage of the pipeline with the outputs of FlawFinder and CodeChecker passing into the orchestration layer.

4.2 Orchestration Layer

Hybrid Testing implements the second stage of Figure 4-1 as a command line tool in Python. The orchestration layer is responsible for parsing the raw output of CodeChecker and FlawFinder to generate the SA-IR. SA-IR standardizes the raw output snippets shown in Figure 4-2 and Table 4.1 to enable both directed fuzzing and manual inspection of results. The orchestration layer is highly configurable, allowing specification of input and output formats, filtering static analysis results by severity, and including or excluding specific code regions by inputting diff files. After generating the SA-IR, the orchestration layer lowers the SA-IR to the desired output format for directed fuzzing.

```
{
    "file": {
        "path": "libxml2/repo/include/libxml/list.h"
     },
     "line": 122,
     "column": 33,
     "message": "'old' declared with a const-qualified typedef type; results in the type being 'struct _xmlList *const' instead of 'const struct _xmlList *'",
     "checker_name": "misc-misplaced-const",
     "severity": "LOW",
     "analyzer_name": "clang-tidy",
     "category": "misc",
     "type": null,
     "bug_path_events": [
         {
             "file": {
                 "path": "libxml2/repo/include/libxml/list.h"
             },
             "line": 24,
             "column": 18,
             "message": "typedef declared here"
         },
         ...
     ],
     "macro_expansions": []
 }
```

Figure 4-2: CodeChecker Output Format

4.2.1 Generating Static Analysis Intermediate Representation

The orchestration layer accepts a static analysis tool name as input to select a parser. CodeChecker requires parsing a JSON file while FlawFinder requires parsing a CSV file. Each parser extracts the fields described in Figure 4-3 from a static analyzer's output to create the SA-IR. Hybrid Testing implements SA-IR as a Python object and generates a new SA-IR for each result produced during static analysis. Initial development showed that the fields in SA-IR were sufficient to provide information to AFLGo [10] and SieveFuzz [70] to begin fuzzing an application.

SA-IR provides flexibility to the Hybrid Testing architecture. By creating an intermedi-

```
class SA_IR:
    file: str = ""
    line: str = ""
    target: str = ""
    checker: str = ""
    category: str = ""
    analyzer: str = ""
    severity: str = ""
    bug: str = ""
    edges: list
```

- file: The relative path to the file containing a potential bug.
- line: The line number where the bug occurs.
- target: The file name without its path, a colon, and the line number ("file:line").
- checker: Which static analyzer module discovered the bug.
- category: Class of bug determined by static analysis (buffer overflow, data race, etc.)
- analyzer: Which analyzer discovered the bug.
- severity: The estimated impact of the bug.
- bug: An optional field used for ground truth testing. See Section 4.4.
- edges: If available, a list of "file:line" targets along the execution path to the bug.

Figure 4-3: Static Analysis Intermediate Representation (SA-IR)

ate representation, this work generalizes all static analyzers to a single format. Integrating an additional static analyzer is simple, requiring only a parser capable of ingesting the analyzer's output. Because static analysis often produces thousands of results with varying severities, the orchestration layer is capable of prioritizing results by command line argument. This prevents data bloat by reducing the number of SA-IR generated and enables this work to evaluate a subset of static analysis targets with potentially high impact. Table 4.2 and Table 4.3 show examples of summary data outputted when parsing SA results.

4.2.2 Lowering SA-IR

After generating the SA-IR, the orchestration layer accepts configuration parameters to specify how the SA-IR should be lowered. During this step, a user can specify a list of source code files to include, limiting the static analysis results to only those occurring in the inputted files. A user can further refine the static analysis results by inputting diff files,

Table 4.2: Orchestration Layer Output — Parsing CodeChecker

Checker Name	Category	Severity	Location
core.uninitialized.Assign	Logic Error	HIGH	xpointer.c:2313
cppcheck-oppositeInnerCondition	Warning	MEDIUM	xmlcatalog.c:136
misc-misplaced-const	Misc	LOW	list.h:122

Table 4.3: Orchestration Layer Output — Parsing FlawFinder

Checker Name	Category	Severity	Location
readlink	race	5	dlfcn.c:466
strcpy	buffer	4	HTMLparser.c:6534
strcat	buffer	4	HTMLparser.c:6535

allowing Hybrid Testing to perform patch testing and target (or avoid) select regions of code. This is necessary because Magma injects bugs by applying a diff to a codebase and Hybrid Testing prioritizes static analysis results within these injected bug regions. Prioritizing and filtering the SA-IR by severity and code location allows Hybrid Testing to first fuzz areas of interest in lieu of sequentially fuzzing all static analysis results.

In practice, specifying a diff file is imperfect. Diff files only include lines of code which changed and static analysis results could occur within some delta of any line present in the diff. The orchestration layer has a configurable parameter for this delta and when a diff is inputted, it performs a bisection search on the lines within the diff to find the closest line to a given SA-IR. If the SA-IR is within the configured delta of the diff, it can then be used as a target location for the desired fuzzer. This configurable delta has two benefits: it allows Hybrid Testing to evaluate potential bugs close to the input diffs (downstream or upstream) and those explicitly within changed code regions.

After the filtering process, the orchestration layer writes the lowered SA-IR to an output file in the format which will be used for fuzzing. AFLGo requires a list of "file:line" pairs to identify which basic blocks to direct fuzzing towards. Initial testing showed that this poses a problem as each SA-IR produces a single "file:line" pair and AFLGo frequently was unable to identify any target basic block when fed a single pair. To mitigate this, the orchestration layer lowers SA-IR to AFLGo by padding the output with a parametrizable number of lines before and after that in the SA-IR. We found that this was sufficient for AFLGo to fuzz an assortment of applications used to evaluate Hybrid Testing.

SieveFuzz, in contrast to AFLGo, requires a specific function name as input to target during fuzzing. Neither CodeChecker nor FlawFinder output the function associated with a static analysis result. To combat this and lower SA-IR to SieveFuzz, Hybrid Testing utilizes GDB's info module to extract the function from a "file:line" pair. While this approach may not work for results occuring in global variables or header files, it was sufficient to allow Hybrid Testing to fuzz a subset of the applications for evaluation. Automatically extracting the function name from a "file:line" pair requires no manual intervention during the testing process, maintaining the fully automated architecture required by Hybrid Testing.

4.3 Directed Fuzzing

Figure 4-1 shows the orchestration layer lowering the SA-IR and beginning the fuzzing process. The orchestration layer requires configuration parameters to specify the directed fuzzer when lowering the SA-IR; this input is parameterized for full automation. Hybrid Testing currently supports two directed fuzzers: AFLGo [10] and SieveFuzz [70]. Both are open-source and have readily available documentation for installation and usage. AFLGo has served as a standard of comparison for many directed fuzzing evaluations and is actively maintained by the open-source community [65, 57, 16, 40, 85, 45, 25, 70, 48]. SieveFuzz is a new directed fuzzer which incorporates static value flow analysis to restrict program state during fuzzing. Directed fuzzers remain an active research technology and are constantly evolving. Hybrid Testing integrates multiple directed fuzzers for the same reason multiple static analyzers are utilized: to reduce the bias and probability that one fuzzer is unable to detect a bug due to inherent limitations of the tool. This work assumes that by expanding the number of fuzzers, the chance of triggering a true positive bug increases. However, the ability of a directed fuzzer to reach and trigger a bug is out of the scope of this work.

In practice, multiple directed fuzzers also increased the number of applications Hybrid Testing could evaluate. AFLGo and SieveFuzz insert checks and logging instrumentation when compiling a codebase. This instrumentation informs input generation and is used to direct fuzzing towards a code region. AFLGo and SieveFuzz are imperfect and frequently fail to instrument codebases, meaning that applications cannot be fuzzed. This work evaluates two directed fuzzers as an initial proof of concept.

4.3.1 AFLGo

When SA-IR is lowered to AFLGo, it produces a series of "file:line" target pairs. AFLGo then performs two compilations: the first attempts to identify basic blocks corresponding to the inputted "file:line" pairs and the second instruments the application based on a distance calculation to the target basic blocks [10]. During development, it was found that AFLGo often failed to identify basic blocks or generate the distance to a basic block. Because of this, there are intermittent static analysis targets which cannot be fuzzed. AFLGo also requires applications to be built with link time optimization (LTO). If an application does not support LTO, Hybrid Testing attempts to use SieveFuzz, which does not require LTO.

Hybrid Testing makes additional enhancements to AFLGo such as recursive detection of intermediate files used for distance calculation (which was previously hardcoded) and the ability to compile applications with assembly only source files. During the evaluation process, we found this necessary to support fuzzing applications within Magma [36]. Despite these enhancements, AFLGo frequently failed to build applications because its distance calculator was unable to identify basic blocks to target based on our static analysis results.

4.3.2 SieveFuzz

When we lower SA-IR to SieveFuzz, it produces a function name where the static analysis result occurs. SieveFuzz utilizes SVF [71] to generate fuzzing test cases and is built on top of a now deprecated version of AFL++ [28]. During the development cycle, we found that applications fuzzable with a non-deprecated AFL++ (such as SQLite3) were unable to be fuzzed by SieveFuzz. Future improvements to Hybrid Testing (and SieveFuzz) could include porting SieveFuzz to a newer version of AFL++. We also found that larger codebases such as OpenSSL caused SieveFuzz to crash, failing to generate any test cases. SieveFuzz was, however, able to fuzz applications AFLGo failed to build such as Lua, making it a valuable integration by expanding the landscape of evaluated applications. In the event that both AFLGo and SieveFuzz are unable to instrument and fuzz an application, Hybrid Testing is currently unable to evaluate it.

To integrate SieveFuzz into Hybrid Testing, it was necessary to remove the job deployment architecture included with SieveFuzz. Section 4.4 describes a containerization policy used instead of this architecture. Additionally, we found that SieveFuzz frequently timed out when evaluating input seeds for fuzzing, even when the same seeds produce no timeouts for AFL or AFL++. If this occurs, we remove the seeds causing the timeouts from an application's seed corpus.

4.4 Magma

This work applies Hybrid Testing to the Magma ground truth fuzzing benchmark [36], integrating AFLGo and SieveFuzz into Magma as well as adding an execution harness to Magma's existing toolset. Magma runs each fuzzer inside a separate Docker [24] container to guarantee cross platform portability and consistency. The integration of AFLGo and SieveFuzz is as described in Section 4.3 where each fuzzer runs in a separate Docker container. Hybrid Testing also integrates CodeChecker and FlawFinder as a "pseudo-fuzzer", meaning that the analyzers are run in exactly the same environment as a fuzzer in the benchmark but perform static analysis in lieu of fuzzing.

Hybrid Testing achieves full automation with Magma via an execution harness which chains together the stages of Figure 4-1. The workflow for Magma first builds a Docker container for a given Magma application and then performs static analysis. The harness then extracts the results from the container, runs the orchestration layer to parse the outputs and generate an SA-IR for each result, and finally signals the orchestration layer to lower the SA-IR to a given fuzzer. Each SA-IR corresponds to a new Docker container which runs the desired directed fuzzer and targets the location specified in the SA-IR. Because static analysis generates many potential results and limited CPU cores were available for the scope of this work, each Docker container is pinned to a CPU core. This enables Hybrid Testing to fuzz many static analysis results in parallel, with the fuzzer responsible for a given result isolated to a single core. While both AFLGo and SieveFuzz are capable of utilizing multiple cores to fuzz an application (and could potentially discover more bugs when doing so), fuzzing many static analysis results in parallel yielded initial results described in Chapter 5.

A key downside to the current Magma containerization policy used by Hybrid Testing is that creating a new Docker container for each SA-IR is storage expensive, requiring an estimated 10 GB of disk space on average per container. Specifying a new lowered SA-IR for each container (and therefore fuzzer) requires repeated instrumentation of an application in

```json
{
    "aflgo": {
        "libxml2": {
            "xmllint": {
                "0": {
                    "reached": {
                        "XML009": 10,
                        "XML012": 10,
                        "XML017": 5,
                        "XML003": 10,
                        "XML006": 10,
                        "XML008": 2705,
                        "XML001": 10
                    },
                    "triggered": {
                        "XML017": 80
                    },
                    "sa_target": "parser.c:2688",
                    "bug": "XML010"
                },
            }
        }
    }
}
```

Figure 4-4: Example Results from Magma with Hybrid Testing — The notation "XML009":10 signifies that AFLGo reached the bug with ID XML009 after 10 seconds of fuzzing.

every container. This can be time intensive as the number of static analysis results scale.

Additionally, Magma uses diff files to inject bugs into the benchmark applications. Hybrid Testing inputs these diff files to the orchestration layer, allowing us to automatically tag an SA-IR with a known bug (see Figure 4-3). This enables the evaluation of static analysis results both within injected bugs and those outside, allowing for true positive and true negative verification. Section 4.2.2 describes how this method is imperfect because diff files in Magma do not always contain bugs (some are instrumentation for logging an already present bug).

Figure 4-4 shows a sample output from fuzzing libxml2. Hybrid Testing augments the output with the static analysis result and the bug corresponding to that result, provided that the static analysis result is within a bug region injected by Magma. This enables the analysis in Chapter 5 and uniquely identifies a fuzzing campaign.

4.5 Discussion

Current limitations of the Hybrid Testing integration with Magma include the repeated creation of Docker [24] containers for each new SA-IR. This significantly hinders performance as each container requires 10 GB of disk space and multiple time-intensive instrumentation stages to compile a Magma application. The majority of disk utilization is derived from custom LLVM installations required for each directed fuzzer. To combat this, Magma could be augmented to use shared memory for each directed fuzzer. The shared memory could be marked readable and executable only, acting as a common layer in each Docker container. This would dramatically reduce the size of each Docker container, enabling higher utilization of compute resources and increasing the maximum number of concurrent fuzzing campaigns.

With respect to the time-intensive instrumentation stages, potential solutions to consolidate the stages are being developed for AFLGo. This could dramatically decrease the time to build an application and begin fuzzing, a process which can take minutes to hours depending on available compute resources. SieveFuzz, on the other hand, recently open-sourced its codebase and has yet to announce any plans to combine instrumentation stages.

Chapter 5

Evaluation

Hybrid Testing is a novel automated testing solution to detect bugs with no false positives and full codebase coverage. It combats the approximately 20% [78] false positive rate of static analysis by using directed fuzzing to validate true positives and eliminate false positives. This work presents initial results using the Magma [36] ground truth fuzzing benchmark and exemplar static analysis tools. The evaluation of Hybrid Testing attempts to answer the following research questions:

- RQ1: Does Hybrid Testing find bugs which are not found through undirected fuzzing (e.g. with AFL), confirming Hybrid Testing can find deeper, complex bugs?
- RQ2a: Can Hybrid Testing validate true positive static analysis reports?
- RQ2b: Can Hybrid Testing eliminate false positives, confirming if a static analysis report is a bug?
- RQ3: Does Hybrid Testing find bugs faster than undirected fuzzing?

To conduct the experiments described in this chapter, we leverage six virtual machines with 32 CPU cores, 64 GB of memory, and 200 GB of physical storage. Each virtual machine has the following CPU specifications:

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   40 bits physical, 48 bits virtual
Byte Order:                      Little Endian
CPU(s):                          32
On-line CPU(s) list:             0-31
Vendor ID:                       GenuineIntel
Model name:                      Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
CPU family:                      6
Model:                           79
Thread(s) per core:              1
Core(s) per socket:              1
Socket(s):                       32
Stepping:                        1
BogoMIPS:                        4399.99
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb
rdtscp lm constant_tsc arch_perfmon rep_good nopl cpuid tsc_known_freq pni
pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm
3dnowprefetch invpcid_single pti ssbd ibrs ibpb tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed
adx smap xsaveopt arat md_clear
Virtualization:                  VT-x
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       1 MiB (32 instances)
L1i cache:                       1 MiB (32 instances)
L2 cache:                        128 MiB (32 instances)
NUMA node(s):                    1
NUMA node0 CPU(s):               0-31
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Mitigation; PTE Inversion; VMX conditional
cache flushes, SMT disabled
Vulnerability Mds:               Mitigation; Clear CPU buffers; SMT Host
state unknown
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Mmio stale data:   Vulnerable: Clear CPU buffers attempted, no
microcode; SMT Host state unknown
Vulnerability Retbleed:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass
disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and
__user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Retpolines, IBPB conditional,
IBRS_FW, STIBP disabled, RSB filling, PBRSB-eIBRS Not affected
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Mitigation; Clear CPU buffers; SMT Host
state unknown
```

It should be noted that this evaluation does not perform parallel fuzzing. The Magma benchmark pins each fuzzing campaign to a single CPU core, disabling parallel fuzzing. The virtual machines used for this evaluation have 32 CPU cores, allowing a maximum of 32 concurrent fuzzing campaigns. Section 4.4 further describes that the current implementation of Hybrid Testing is not capable of fully utilizing all 32 cores due to limited physical storage, supporting an average of 24 concurrent fuzzing campaigns per virtual machine.

5.1 Methodology

We leverage the Magma fuzzing benchmark [36] for evaluation. Magma consists of multiple open-source applications with injected bugs from real world vulnerabilities, making it an ideal candidate to show Hybrid Testing is capable of detecting true and false positives.

In total, Hybrid Testing was able to evaluate six of the nine benchmarks included in Magma. This includes libpng, libsndfile, libtiff, libxml2, lua, and poppler. Hybrid Testing was unable to evaluate openssl, php, and sqlite3 because both AFLGo and SieveFuzz failed to build and instrument the Magma applications. It should also be noted that the experiments conducted using Magma were performed with a single fuzzing harness per application. Applications such as libxml2, poppler, and libtiff provide multiple fuzzing harnesses which can be used to fuzz different parts of the application. The initial results shown in this chapter use a single fuzzing harness per application due to the limited CPU power available during the course of this work.

Throughout the course of this chapter, we use bug severity and location as experimental parameters to setup our evaluation of Hybrid Testing. This allows us to performs three classes of evaluation using Magma to answer our research questions:

- Hybrid Testing fed with static analysis results deemed high severity and are within the injected bug regions
- Hybrid Testing fed with static analysis results deemed medium or higher severity and are within the injected bug regions
- Hybrid Testing fed with static analysis results deemed high severity and are outside of the injected bug regions

Splitting the evaluation by static analysis report severity allows us to prioritize fuzzing SA results of higher severity. We prioritize higher severity results because we assume that these SA reports are more likely to be true positive bugs. As future work, Hybrid Testing should be evaluated using all SA reports, regardless of severity. For the remainder of the evaluation, we compare Hybrid Testing's ability to detect more bugs, validate true and false positives, and detect bugs faster across different severities of static analysis warnings.

This work also splits the evaluation by static analysis results within the bugs injected by Magma and those outside the injected bug regions. This is accomplished by the method described in Section 4.2.2. Splitting the evaluation along this axis allows confirmation of true positives and elimination of false positives. If a static analysis result is within an injected bug region, we fuzz the result using Hybrid Testing and detect it through Magma's Ideal Sanitizer instead of AFLGo or SieveFuzz's Address Sanitizer. We chose to use the Ideal Sanitizer for Magma injected bugs as it produces a crash any time an injected bug is triggered and has low overhead. Without the Ideal Sanitizer, a Magma injected bug may not cause a crash. If a static analysis result is outside an injected bug region, we again use Hybrid Testing but instead detect any bugs using an address sanitizer and crashing inputs found through fuzzing. We chose to use an address sanitizer because it can detect a wide class of bugs and must only be run for crashing inputs found through fuzzing as part of our root cause analysis described in RQ2b. Splitting the evaluation along this axis allows Hybrid Testing to evaluate scenarios where ground truth is known (i.e. the injected bug regions) and scenarios where ground truth is unknown.

As a baseline of comparison, this work pits Hybrid Testing against AFL, a long-standing, undirected fuzzer supported by Magma. Comparing against AFL enables analysis of Hybrid Testing's performance with respect to current testing tools and its ability to find different bugs or detect bugs faster. This is an important step in demonstrating that Hybrid Testing is an improvement to current automated testing solutions.

To answer RQ1, we feed Hybrid Testing static analysis results inside the bugs injected by Magma. We examine if fuzzing SA results inside the injected bug regions detects more of Magma's constituent bugs. We compare the number of bugs detected per application fuzzed by Hybrid Testing to a baseline of AFL.

To answer RQ2a, we compare Hybrid Testing's ability to fuzz and trigger an injected bug at a given static analysis location. This allows us to evaluate how well Hybrid Testing can detect true positives when ground truth data is available (in the form of an injected bug). To answer RQ2b, if Hybrid Testing eliminates false positives, we feed Hybrid Testing SA reports outside of Magma's injected bugs and examine if any crashes detectable with address sanitization (ASan) are found.

To answer RQ3, we feed Hybrid Testing static analysis results inside of Magma's injected bug regions. Magma's instrumentation allows us to capture the time to reach and trigger a crash for each injected bug. This enables our performance analysis against AFL.

Each class of evaluation, and the baseline of AFL, was performed with adherence to fuzzing practices suggested by *Evaluating fuzz testing* [44]. Fuzzing results can vary dramatically across runs and as the time spent fuzzing changes. This work runs each evaluation class for 24 hours. We compute the mean time to reach and trigger ground truth bugs across three such runs. We also present the total unique bugs detected across all three runs.

5.2 Static Analysis Results

We evaluate Hybrid Testing across two experimental parameters, severity of static analysis results and if a result is within a bug injected by Magma. Table 5.1 shows the number of SA results for each application included in the Magma benchmark. The table includes results from both CodeChecker [27] and FlawFinder [29]. We elected to combine results from both static analyzers to increase the total number of bugs detected by Hybrid Testing. The columns marked injected correspond to static analysis results within the bugs injected by Magma and the severity corresponds to the minimum threshold a static analysis result must have to be included in the count. The table shows that as the severity threshold decreases, the number of results increases. This is consistent with previous evaluations of static analyzers [4, 22]. In Table 5.1, we also show the number of bugs injected by Magma and the number of bugs we failed to detect with static analysis.

Table 5.1: Magma Static Analysis Results Count — The counts shown in this table include results from both CodeChecker and FlawFinder which we ran in parallel. The number of bugs not detected are bugs injected by Magma which Hybrid Testing did not detect with static analysis.

Target	High Severity — Injected	Medium Severity — Injected	High Severity	Medium Severity	Total Injected Bugs	Injected Bugs Not Detected
libpng	1	5	35	490	7	5
libsndfile	2	15	21	370	25	15
libtiff	7	22	29	770	14	8
libxml2	0	13	35	906	17	11
lua	1	8	46	222	4	1
poppler	1	14	202	1100	22	17

This work evaluates static analysis results with high and medium severities. However, future evaluation of Hybrid Testing should attempt to test all results detected via static analysis. It is possible that even though an injected bug exists, a static analyzer could deem it low severity.

Table 5.3 shows a further breakdown of static analysis results by the injected bug corresponding to a SA result. It should again be noted that high severity results are a subset of the medium severity results. Hybrid Testing uses the severity as a threshold for what should

Table 5.2: Injected Bugs in Magma Applications and Bugs Found with Static Analysis

Magma Application	Injected Bugs	Injected Bugs Detected by SA
libpng	7	2
libsndfile	25	10
libtiff	14	6
libxml2	17	6
lua	4	3
poppler	22	5

be fuzzed. Setting a medium threshold allows Hybrid Testing to capture static analysis targets with a minimum severity that correspond to the same bug. For example, TIF001 has six high severity threshold results and 12 medium severity threshold results shown in Table 5.3. This decision was made to prioritize bugs with more static analysis results, fuzzing different targets within the same bug.

Table 5.2 complements this data, showing that static analysis did not detect every injected bug. Static analysis detected 29% of bugs in lua, 40% in libsndfile, 43% in libtiff, 35% in libxml2, 75% in lua, and 23% in poppler. The number of injected bugs detected is discussed further in Section 5.4 as part of Hybrid Testing's false negative rate.

During the evaluation, some SA targets were unable to be instrumented and thus are excluded from the evaluation. The number of targets which were unable to be fuzzed are shown in Table 5.5. In total 61% of libpng, 11% of libsndfile, 36% of libtiff, 23% of libxml2, 87% of lua, and 82% of poppler static analysis results were unable to be instrumented. The failure to instrument many of these targets can be explained by multiple scenarios: the algorithm used by a directed fuzzer to perform instrumentation fails to identify the SA target (frequently the case with AFLGo's distance calculator) or the fuzzer imposes unsupported requirements such as LTO capabilities on a codebase. Further work on filtering which static analysis targets to instrument and fuzz is discussed in Chapter 6.

5.3 RQ1 — Bug Detectability

A primary goal of Hybrid Testing is to improve upon traditional fuzzing techniques by finding new or different bugs. Table 5.4 shows the bugs found by Hybrid Testing within Magma. The table is broken down by each evaluation class described in Section 5.1 and shows the

Table 5.3: Breakdown of Static Analysis Targets by Bug — The table shows only Magma injected bugs with one static analysis result or more.

BugID	Severity	SA Targets from Bug
TIF012	Medium	1
TIF007	Medium	2
TIF004	Medium	1
TIF002	Medium	2
TIF001	Medium	12
TIF003	Medium	1
TIF001	High	6
TIF003	High	1
SND005	Medium	1
SND015	Medium	2
SND014	Medium	2
SND010	Medium	1
SND012	Medium	2
SND013	Medium	3
SND004	Medium	1
SND002	Medium	1
SND025	Medium	1
SND017	Medium	1
SND004	High	1
SND025	High	1
PNG004	Medium	1
PNG007	Medium	1
PNG004	High	1
PDF018	Medium	1
PDF008	Medium	2
PDF003	Medium	4
PDF002	Medium	1
PDF012	Medium	1
PDF008	High	1
XML010	Medium	1
XML008	Medium	1
XML003	Medium	1
XML006	Medium	5
XML005	Medium	2
XML011	Medium	3
LUA002	Medium	1
LUA004	Medium	4
LUA003	Medium	3
LUA004	High	1

Table 5.4: Unique Bugs Triggered by Each Fuzzing Evaluation Class — This table shows only Magma applications which were successfully evaluated using Hybrid Testing.

Fuzzer	Magma Application	BugID
afl	libpng	PNG003,PNG007
afl	libsndfile	SND005,SND017
afl	libtiff	TIF007,TIF009,TIF012,TIF014
afl	libxml2	XML017
afl	lua	LUA004
afl	poppler	PDF011,PDF016
aflgo_inj_high	libpng	PNG003,PNG007
aflgo_inj_high	libsndfile	SND005,SND017
aflgo_inj_high	libtiff	TIF006,TIF007,TIF009,TIF012,TIF014
aflgo_inj_high	poppler	PDF011,PDF016
aflgo_inj_medium	libpng	PNG003,PNG007
aflgo_inj_medium	libsndfile	SND005,SND017
aflgo_inj_medium	libtiff	TIF006,TIF007,TIF009,TIF012,TIF014
aflgo_inj_medium	libxml2	XML017
aflgo_inj_medium	poppler	PDF011,PDF016
sievefuzz_inj_high	lua	LUA004
sievefuzz_inj_medium	lua	LUA004,LUA003

In the this table, the suffix "inj" signifies that the static analysis targets fed to the fuzzer were located within the bugs injected by Magma. The suffixes "high" and "medium" signify that the static analysis targets were given that severity or higher by the static analyzer.

unique bugs triggered for each class and Magma application. The table shows that Hybrid Testing finds every bug found by AFL. Hybrid Testing also finds two bugs which AFL does not, TIF006 and LUA003. AFL detected 12 total bugs while Hybrid Testing found 14. The table shows that the bug LUA003 is only detected when fuzzing static analysis results of medium severity or higher. Figure 5-1 complements this data, showing Hybrid Testing performs at least as well as AFL when fed with SA targets inside Magma's injected bugs.

Hybrid Testing finds more bugs, including those AFL finds as depicted by Table 5.4. Hybrid Testing is an improvement to existing techniques, demonstrating a 17% increase in total bug detection across all applications in Magma. It should also be noted that Hybrid Testing did not detect all Magma bugs with static analysis as shown in Table 5.2. Integrating additional static analyzers or lowering the severity threshold of flagged SA reports could allow us to fuzz and detect more of Magma's injected bugs. Detecting more bugs is a crucial step towards more secure software and Hybrid Testing has proven it can aid in this detection.

Table 5.5: Breakdown of Static Analysis Targets Unable to be Fuzzed by Hybrid Testing

Fuzzer	Magma Application	Number of SA Targets that Failed to Build
aflgo_inj_medium	libtiff	3
aflgo_high	libtiff	17
aflgo_inj_high	libtiff	1
aflgo_inj_medium	libsndfile	0
aflgo_high	libsndfile	4
aflgo_inj_high	libsndfile	0
aflgo_inj_medium	libpng	3
aflgo_high	libpng	22
aflgo_inj_high	libpng	0
aflgo_inj_medium	poppler	5
aflgo_high	poppler	173
aflgo_inj_high	poppler	0
aflgo_inj_medium	libxml2	0
aflgo_high	libxml2	11
sievefuzz_inj_medium	lua	7
sievefuzz_high	lua	41
sievefuzz_inj_high	lua	0

In the this table, the suffix "inj" signifies that the static analysis targets fed to the fuzzer were located within the bugs injected by Magma. The suffixes "high" and "medium" signify that the static analysis targets were given that severity or higher by the static analyzer.

RQ1 Key Takeaways:

- Hybrid Testing finds a strict super set of the all Magma bugs found by AFL
- Hybrid Testing found 17% more bugs than AFL when fuzzing the Magma suite

5.4 RQ2a — True Positive Verification

To evaluate Hybrid Testing's ability to validate true positives, this work examines scenarios where a static analysis target is within a bug injected by Magma. We compare the bug found by static analysis to the bugs triggered when directed fuzzing to determine if a SA target is a true positive. Section 4.4 discusses how we augment Magma's output with the bug and SA target used for a given fuzzing campaign. We use this comparison to generate the data for this portion of the evaluation.

Figure 5-2 shows the percentage of static analysis results confirmed as true positives by

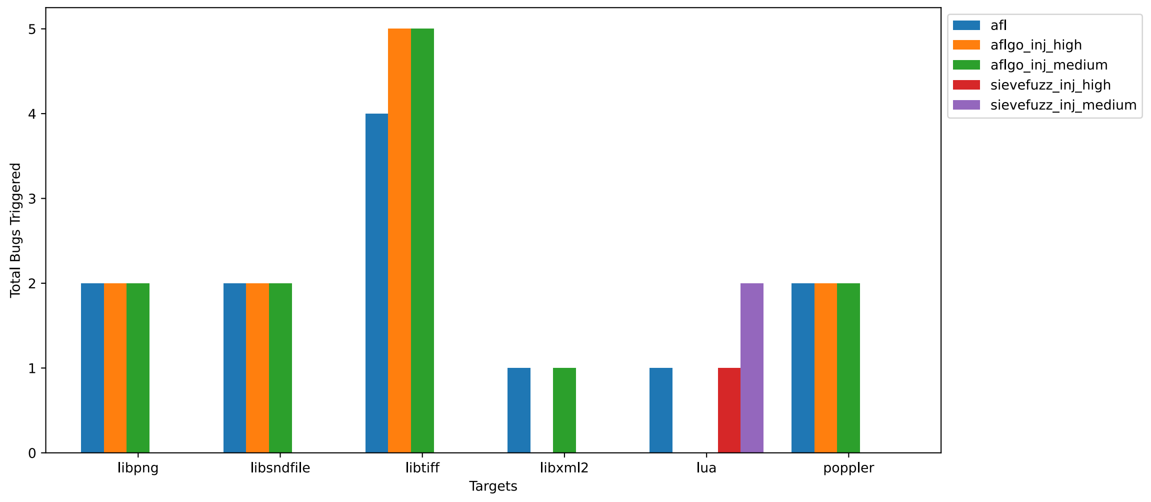

Figure 5-1: Fuzzing Medium and High Severity Static Analysis Targets Inside of Injected Bugs vs. AFL

Hybrid Testing. The percentages shown are reflective of static analysis targets which were successfully instrumented by the directed fuzzer. While it is possible that other injected bugs may be triggered while attempting to verify a true positive, this portion of the evaluation focuses only on scenarios where a directed fuzzer reached the location it was attempting to (as specified by the SA-IR), triggering a bug.

When fuzzing medium severity SA targets, Hybrid Testing had a 50% true positive rate for libpng, 20% for libsndfile, and 33% for libtiff. Hybrid Testing did not find any true positives for libxml2 and poppler, but did have a 67% true positive rate for lua. The results in Figure 5-2 show that static analysis targets corresponding to true positive bugs are frequently marked as medium severity, consistent with Table 5.3.

Table 5.6: Hybrid Testing False Negative Rate

Magma Application	Injected Bugs	Injected Bugs Detected by SA
libpng	7	2
libsndfile	25	10
libtiff	14	6
libxml2	17	6
lua	4	3
poppler	22	5

The percentage of true positives detected in Figure 5-2 could be explained partially

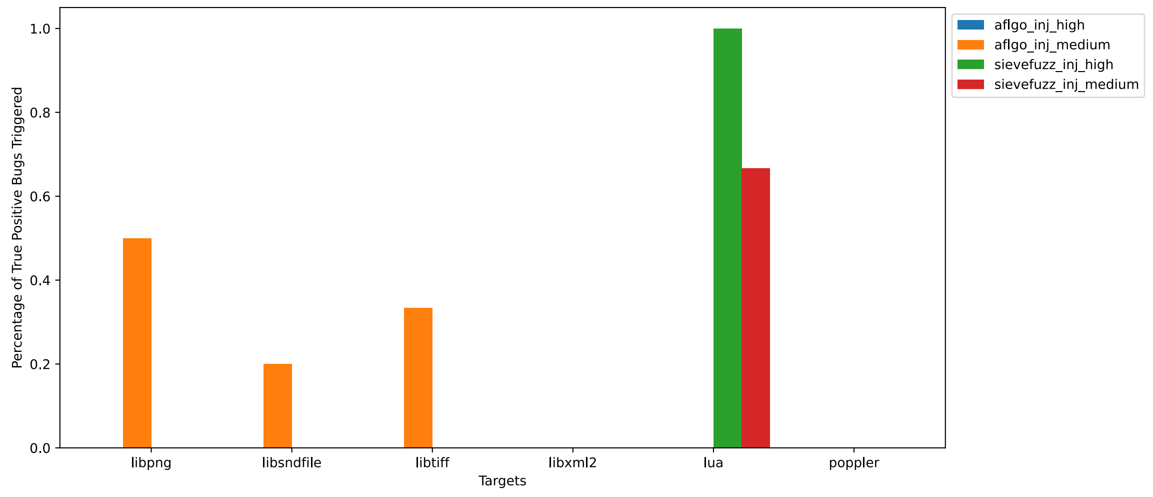

Figure 5-2: Percentage of Static Analysis Results Verified as True Positives by Hybrid Testing — The percentage of true positive bugs is calculated by taking the number of Magma injected bugs detected through fuzzing divided by the total number of injected bugs Hybrid Testing identified with static analysis.

because of the number of injected bugs Hybrid Testing found with static analysis. Table 5.2 shows that Hybrid Testing's static analysis stage did not detect five bugs in libpng, 15 in libsndfile, eight in libtiff, 11 in libxml2, one in lua, and 17 in poppler. Table 5.10 provides further insight, showcasing that Hybrid Testing was able to reach many of the injected bugs but did not satisfy the conditions to trigger a crash.

The results in this section are impacted by Hybrid Testing's inability to detect many bugs with static analysis as shown in Table 5.2. If Hybrid Testing does not detect a bug with SA, it cannot fuzz a code region. Of the bugs Hybrid Testing did detect with SA, it triggered an average of 28% of those bugs. This corresponds to an average of 14% of all bugs injected by Magma, which is again heavily impacted by the limitations of static analysis. In comparison, AFL verified 18% of all Magma's injected bugs. This makes sense given that it is undirected and can fuzz any code location regardless of SA results.

The other side of Hybrid Testing's true positive rate is the false negative rate. We define false negatives as bugs injected by Magma into an application, but not detected by static analysis. If no static analysis target corresponds to a bug, Hybrid Testing is unable to fuzz it and is thus a false negative. Table 5.6 shows the number of bugs injected by Magma into each application as well as the number of unique bugs detected by static analysis. This table shows that libpng has a false negative rate of 71%, libsndfile 60%, libtiff 57%, libxml2

65%, lua 25%, and poppler 77%. This false negative rate could be reduced by integrating additional static analyzers as well as lowering the severity threshold for allowed SA targets. The static analyzers have been tuned to reduce false positives given the high and medium severity thresholds, which is likely increasing false negatives. In the future, Hybrid Testing could act as a better triaging tool for future work on more sensitive static analyzers.

Figure 5-2 shows that Hybrid Testing failed to validate any SA targets as true positives for both libxml2 and poppler. This could be explained by two potential scenarios: the bugs take longer than 24 hours of fuzzing to detect or the bugs are deeper in the codebase, requiring passing through other injected Magma bugs. This especially holds true when examining Table 5.10 for bugs in libxml2 (prefix XML) and bugs in poppler (prefix PDF). The XML and PDF bugs are often triggered within seconds of beginning fuzzing, implying that they are likely shallow bugs. If a static analysis result occurred deeper within the codebase, it is possible that it must pass through a shallow bug to reach that static analysis result. Such shallow bugs could be blocking fuzzing from reaching further into the codebase, preventing detection with Hybrid Testing. It is possible that directed fuzzing is not able to find a path to a deeper, more complex bug if it must first pass through a shallow bug.

Table 5.10 also shows that in many cases, directed fuzzing was able to reach but not trigger bugs for libxml2 and poppler. Magma defines reaching as arriving at the lines of code where a bug exists but not satisfying the input conditions to trigger the bug. Fuzzing all of the applications for a longer duration could increase the number of true positives Hybrid Testing is able to verify. Future work could include symbolizing the generated test cases which reach an injected bug but do not trigger it. This would allow us to examine if there exists a data flow which would trigger the bug along the execution path that reached the injected bug instrumentation. Analysis of this data flow could then corroborate if more fuzzing time is needed to achieve the correct variable values to trigger the bug.

RQ2a Key Takeaways:

- Hybrid Testing verified an average of 28% of the 32 bugs detected by static analysis across all applications in the Magma suite
- Hybrid Testing successfully instrumented and evaluated 59 of the 77 static analysis reports corresponding to Magma bugs
- Hybrid Testing verified an average of 14% of all bugs injected by Magma while AFL verified 18% of all injected bugs
- Hybrid Testing has an average false negative rate of 59% for Magma injected bugs

5.5 RQ2b — False Positive Elimination

A crucial component of Hybrid Testing is its ability to detect false positives and correctly classify these false positive reports as true negatives. One of the greatest shortcomings of static analysis is the high false positive rate, frequently causing developers to ignore tool warnings and decreasing software security [63, 23].

We classify the static analysis targets within bugs injected by Magma as true positives and preliminarily classify those outside of Magma as false positives. We then use Hybrid Testing to validate if the SA targets outside the injected bugs are false positives. To eliminate false positives and verify that a static analysis result is a true negative, Hybrid Testing fuzzes a SA target for 24 hours. If any crashes are detected during this fuzzing campaign, the Magma application is built with ASan and the location of the crash is compared against the SA target fed to the directed fuzzer. If no crashes are found at the location of the static analysis target, the result is deemed a false positive and is thus a true negative. The limitations to this approach are discussed in Chapter 6.

In this section, we analyze Hybrid Testing's ability to find vulnerabilities when ground truth data is unavailable . We evaluate if Hybrid Testing detects additional bugs when using SA targets to inform directed fuzzing. We compare against AFL to determine if undirected fuzzing finds the same set of vulnerabilities as Hybrid Testing. This comparison allows us to evalaute if Hybrid Testing is an improvement to current automated testing practices.

Table 5.7 shows a breakdown of crashes by fuzzer and Magma application. The figure reports the number of crashing inputs found during fuzzing and a classification for the crash. Libpng reported 140 crashes and lua reported 12 crashes when using AFL, however these were due to the memory limit of 100 MB imposed on each fuzzer test case. To validate this, we ran the crashing inputs for libpng and lua on binaries built with address sanitization without memory limits and no crashes were detected. The same memory limit issue occurred when fuzzing libpng with AFLGo. Future work should run all fuzzing experiments again with a memory limit greater than 100 MB to capture memory intensive bugs.

Table 5.7 also shows that libxml2 and poppler, when fuzzed with AFL, reported 1405 and 573 crashes respectively. Poppler fuzzed with AFLGo reported 110 crashes and libxml2 reported 0. We ran each crashing test case with address sanitization and no errors were reported. Upon closer inspection of libxml2's and poppler's output logs, these crashes were detected as syntax errors and warnings by each application; libxml2 and poppler gracefully exited after detecting malformed inputs. There was also one such graceful exit when fuzzing libsndfile with AFL. To combat this, Hybrid Testing could attempt to detect such graceful exits and not report them as crashes.

We inspected the summary reports generated by ASan for libsndfile and libtiff using AFL's crashing test cases, as well as libtiff for AFLGo. Table 5.7 shows that we found two unique crashing lines of code for libsndfile and two unique lines for libtiff using the test cases generated by AFL. Table 5.8 shows that the crashing lines for libsndfile corresponded to the Magma injected bugs SND005 and SND017. The two unique crashing lines found in libtiff by AFL also corresponded to TIF009. Given that AFL is undirected, it likely explored execution paths containing injected bugs. AFLGo, on the other hand, was directed towards static analysis targets outside of those injected by Magma which could be why it did not trigger bugs in libsndfile.

When we analyzed the ASan summary reports for libtiff generated by AFLGo's crashing test cases, we found four unique crashing lines of code listed in Table 5.8. Two of these lines were identical to those found by AFL, again corresponding to the injected bug TIF009. Additionally, AFLGo found two lines that are direct results of CVE-2022-4645, which exists outside of the Magma benchmark [20, 21]. However, cross referencing the four crashing lines with the static analysis targets fed to AFLGo showed that no crashes occurred at locations AFLGo attempted to direct fuzzing towards. This could be because the static analysis target was deeper into the codebase than where a bug occurred, meaning that while trying to reach a static analysis target the fuzzer triggered a bug along the execution path to that target. In this case, CVE-2022-4645 could be blocking AFLGo from reaching deeper into the codebase to a given static analysis target. Future evaluation should attempt to patch this bug and fuzz libtiff again to analyze if directed fuzzing was able explore deeper into the source code.

Table 5.7: Unique Bugs Detected with ASan

Fuzzer	Magma Application	Graceful Exits	Crashes due to Memory Limit	Crashes Triggering ASan	Unique Crash Locations
afl	libpng	0	140	0	0
afl	libsndfile	1	0	33	2
afl	libtiff	0	0	13	2
afl	libxml2	1405	0	0	0
afl	lua	0	12	0	0
afl	poppler	573	0	0	0
aflgo_high	libpng	0	109	0	0
aflgo_high	libsndfile	0	0	0	0
aflgo_high	libtiff	0	0	180	4
aflgo_high	libxml2	0	0	0	0
aflgo_high	poppler	110	0	0	0
sievefuzz_high	lua	0	0	0	0

In the this table, the suffix "high" signifies that the static analysis targets were given that severity by the static analyzer.

Table 5.8: Hybrid Testing ASan Crashes

Fuzzer	Magma Application	Crashing Line	Cause
afl	libsndfile	aiff.c:1790	SND005
afl	libsndfile	wavlike.c:353	SND017
afl	libtiff	tif_dirwrite.c:2111	TIF009
afl	libtiff	tif_dirwrite.c:2124	TIF009
aflgo_high	libtiff	tif_dirwrite.c:2111	TIF009
aflgo_high	libtiff	tif_dirwrite.c:2124	TIF009
aflgo_high	libtiff	tiffcp.c:948	CVE-2022-4645
aflgo_high	libtiff	tif_dir.c:498	CVE-2022-4645

Table 5.9 shows a breakdown of the high severity SA targets for each application evaluated within the Magma benchmark. The number of high severity SA targets is computed by subtracting the targets which failed to be instrumented in Table 5.5 from the high severity targets (outside of Magma injected bugs) in Table 5.1. The number of SA targets validated is the number of high severity targets Hybrid Testing verified with address sanitization.

Table 5.9: Hybrid Testing High Severity Analysis — Only the static analysis targets which were successfully instrumented and evaluated are shown.

Fuzzer	Magma Application	High Severity SA Targets	SA Targets Validated
aflgo	libpng	13	0
aflgo	libsndfile	17	0
aflgo	libtiff	12	0
aflgo	libxml2	24	0
sievefuzz	lua	5	0
aflgo	poppler	29	0

Of the high severity SA targets we evaluated, Hybrid Testing did not detect any crashes at the lines of code flagged by static analysis. This is primarily a symptom of the directed fuzzer's inability to instrument a codebase for a given SA target. Table 5.5 again shows that we were unable to evaluate the majority of high severity SA results. While this would mean that all SA targets shown in Table 5.9 are false positives, this is not representative of the underlying static analyzers due to the failed instrumentation. Hybrid Testing should integrate additional directed fuzzers to increase its ability to instrument and evaluate targets. Additional time should be spent fuzzing each Magma application to catch deeper bugs which are not detectable within 24 hours of fuzzing.

RQ2b Key Takeaways:

- Hybrid Testing marked all high severity SA targets as false positives, primarily due to the inability of a directed fuzzer to instrument the majority of SA targets
- Hybrid Testing should incorporate additional directed fuzzers and fuzz applications for more than 24 hours to increase the number of SA targets we can instrument and validate

5.6 RQ3 — Performance Analysis and Time to Detection

A secondary goal of this work was to evaluate how efficiently Hybrid Testing finds bugs in comparison to current techniques, namely fuzzing with AFL. Table 5.10 shows a breakdown of how long it took to reach and trigger a bug injected by Magma. To reach a bug, a fuzzer must find an input that arrives at the line of code where a bug was injected. To trigger said bug, the input must satisfy a constraint when it reaches that line of code, causing a crash. The figure shows that AFLGo was generally slower to detect and trigger a bug, while SieveFuzz was faster in every scenario. Table A.1 complements this conclusion, showing the standard error and average time in seconds to reach each Magma bug. Hybrid Testing beats AFL for five bugs in libsndfile, one in libpng, three in libtiff, one in poppler, and two in lua. For bugs in both libtiff and lua, Hybrid Testing triggered bugs AFL was unable to detect (TIF006, LUA003) and reached the same bugs in all applications as AFL, except PDF008 in poppler.

Table 5.10: Time to Reach and Trigger: Static Analysis Targets Inside of Injected Bugs and AFL — In the table header, R represents the time to reach a bug, T represents the time to trigger said bug, and Δ represents the difference in time to trigger a bug versus AFL. The suffix I.M. signifies Injected Medium, I.H. signifies Injected High, and SF represents SieveFuzz. In the table data, X signifies that Hybrid Testing was unable to fuzz the application where a given bug occurs while '-' signifies that fuzzing did not reach or trigger said bug. A positive Δ indicates it took more time than AFL and a negative Δ represents less time. A Δ of '-' means that AFL did not trigger the bug.

Fuzzer	AFL			AFLGo I.M.			AFLGo I.H			SF I.M.			SF I.H.		
Metric	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ
Bug ID															
PNG003	5s	10s	0s	5s	10s	0s	5s	10s	0s	X	X	X	X	X	X
PDF016	5s	35s	0s	5s	2m	+2m	5s	1m	+1m	X	X	X	X	X	X
SND005	5s	17m	0s	5s	22m	+5m	5s	30m	+13m	X	X	X	X	X	X
SND017	6m	50m	0s	8m	1h	+10m	15m	1h	+10m	X	X	X	X	X	X
TIF007	13m	2h	0s	14m	4h	+2h	16m	5h	+3h	X	X	X	X	X	X
PDF011	10s	3h	0s	10s	10h	+7h	10s	2h	-1h	X	X	X	X	X	X
XML017	5s	1m	0s	5s	1m	0s	-	-	-	X	X	X	X	X	X
TIF009	9h	9h	0s	14h	14h	+5h	12h	12h	+3h	X	X	X	X	X	X
TIF012	5s	10h	0s	5s	12h	+2h	5s	14h	+4h	X	X	X	X	X	X
LUA004	16h	16h	0s	X	X	X	X	X	X	12h	12h	-4h	14h	14h	-2h
TIF014	13m	16h	0s	14m	17h	+1h	16m	19h	+3h	X	X	X	X	X	X
PNG007	10s	21h	0s	10s	22h	+1h	10s	15h	-6h	X	X	X	X	X	X
TIF006	-	-	0s	18h	18h	-	18h	18h	-	X	X	X	X	X	X

Continued on next page

Fuzzer	AFL			AFLGo I.M.			AFLGo I.H			SF I.M.			SF I.H		
Metric	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ
Bug ID															
LUA003	-	-	0s	X	X	X	X	X	X	1m	23h	-	1m	-	-
TIF002	-	-	0s	18h	-	-	18h	-	-	X	X	X	X	X	X
TIF003	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
TIF008	-	-	0s	-	-	-	23h	-	-	X	X	X	X	X	X
TIF010	9h	-	0s	10h	-	-	12h	-	-	X	X	X	X	X	X
XML001	10s	-	0s	10s	-	-	-	-	-	X	X	X	X	X	X
XML003	10s	-	0s	10s	-	-	-	-	-	X	X	X	X	X	X
XML006	10s	-	0s	10s	-	-	-	-	-	X	X	X	X	X	X
XML008	27m	-	0s	40m	-	-	-	-	-	X	X	X	X	X	X
XML009	10s	-	0s	10s	-	-	-	-	-	X	X	X	X	X	X
SND024	1m	-	0s	21s	-	-	10s	-	-	X	X	X	X	X	X
SND020	7m	-	0s	12m	-	-	16m	-	-	X	X	X	X	X	X
PDF009	10s	-	0s	11s	-	-	12s	-	-	X	X	X	X	X	X
SND016	1m	-	0s	21s	-	-	10s	-	-	X	X	X	X	X	X
PDF007	15s	-	0s	16s	-	-	17s	-	-	X	X	X	X	X	X
PDF012	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
PDF014	10s	-	0s	15s	-	-	15s	-	-	X	X	X	X	X	X
PDF005	17s	-	0s	6h	-	-	16h	-	-	X	X	X	X	X	X
PDF019	25s	-	0s	29s	-	-	33s	-	-	X	X	X	X	X	X
PDF021	15s	-	0s	16s	-	-	17s	-	-	X	X	X	X	X	X
PNG001	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
PDF003	12s	-	0s	15s	-	-	17s	-	-	X	X	X	X	X	X
PNG004	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
PNG005	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
PNG006	10s	-	0s	10s	-	-	10s	-	-	X	X	X	X	X	X
SND001	1m	-	0s	21s	-	-	10s	-	-	X	X	X	X	X	X
XML012	10s	-	0s	10s	-	-	-	-	-	X	X	X	X	X	X
SND006	1m	-	0s	21s	-	-	10s	-	-	X	X	X	X	X	X

Continued on next page

Fuzzer	AFL			AFLGo I.M.			AFLGo I.H			SF I.M.			SF I.H		
Metric	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ	R	T	Δ
Bug ID															
SND007	1m	-	0s	21s	-	-	10s	-	-	X	X	X	X	X	X
PDF008	20s	-	0s	-	-	-	-	-	-	X	X	X	X	X	X
PDF002	17s	-	0s	6h	-	-	16h	-	-	X	X	X	X	X	X

Table 5.11 shows the average time to reach and trigger a bug for each application in the Magma suite. The table is broken down by fuzzer and is computed across three fuzzing runs of 24 hours, with the standard error for each average shown. The table shows that Hybrid Testing (when using SieveFuzz) was 66% faster to reach and 25% faster to trigger bugs in lua. Hybrid Testing was 25% faster to trigger a bug than AFL when fuzzing poppler with high severity SA targets; however, it was 240% slower to trigger bugs when fed with SA targets given a medium severity threshold. This is likely explained by PDF011, which is triggered in two hours with high severity targets, but 10 hours with a medium severity threshold. Table A.1 shows the standard error to reach and trigger PDF011, indicating that it can vary two to six hours across fuzzing runs. The difference between trigger times for high and medium severity thresholds could be due to directed fuzzing inadvertently triggering PDF011 while trying to reach deeper into the codebase to fuzz medium severity targets.

Table 5.11: Average Time to Reach and Trigger a Magma Bug with Standard Error — The average time to a bug is computed across three fuzzing runs of 24 hours. The standard error is also computed across three such runs. Only targets which were successfully evaluated with Hybrid Testing are shown.

Fuzzer	Target	Reach Time Avg. (s)	Reach Time Std. Err. (s)	Trigger Time Avg. (s)	Trigger Time Std. Err. (s)
afl	libpng	9.2	1.9	15312.5	26504.7
afl	libsndfile	151.5	160.1	2028.3	1585.6
afl	libtiff	11636.9	15980.9	35405.4	17234.5
afl	libxml2	244.8	692	90	21.2
afl	lua	60123.3	657.2	60123.3	657.2
afl	poppler	13.8	5.3	5757.5	5722.5
aflgo_inj_high	libpng	9.2	1.9	15466	20744.1
aflgo_inj_high	libsndfile	249	586.5	3455.8	2360.2
aflgo_inj_high	libtiff	14463.6	20233.1	41326	20129.2
aflgo_inj_high	poppler	15.3	7.6	4324.2	6797.1
aflgo_inj_medium	libpng	9.2	1.9	24112.2	34550
aflgo_inj_medium	libsndfile	169.2	417.8	2888.7	1983.3
aflgo_inj_medium	libtiff	15857.5	22147.4	39934.4	20819
aflgo_inj_medium	libxml2	355.3	975.3	107.2	49.9
aflgo_inj_medium	poppler	15	6.5	19538.6	25043.6
sievefuzz_inj_high	lua	25750	25680.3	51430	169.5
sievefuzz_inj_medium	lua	20208.9	22738.5	45347.7	14877.4

With respect to cases where AFLGo is marginally slower than AFL, this could be explained by their similar underlying architecture. AFLGo is built on top of AFL, with additional calculations and instrumentation feedback for directing execution towards a target location. For example, PDF016 is triggered in 35 seconds by AFL and two minutes by AFLGo. The difference in trigger time is small and could be explained by inherent randomness during the input generation process. However, AFLGo does realize advantages for PDF011 and PNG007, triggering both bugs over an hour faster than AFL. Table A.1 shows that the standard error to trigger PDF011 varies two to six hours across different static analysis severity thresholds. Table A.1 also shows that the error to trigger PNG007 varies between three to four hours across SA target severities. While the efficiency of directed fuzzing is out of the scope of this work, Hybrid Testing does show that it can offer performance improvements over current testing tools.

Though SieveFuzz was only capable of evaluating lua, it performed substantially better than AFL. For LUA004, SieveFuzz found the bug over four hours faster than AFL. For LUA003, AFL was unable to reach or trigger the bug while SieveFuzz accomplished both. The standard error for reaching and triggering both bugs is shown in Table A.1. Performance gains in SieveFuzz could be attributed to multiple factors. It is built on top of AFL++, which has been shown to outperform AFL [28]. SieveFuzz also uses a novel state restriction algorithm, potentially allowing it to better direct fuzzing towards a target location. Hybrid Testing again outperforms traditional, undirected fuzzing when we use SieveFuzz.

RQ3 Key Takeaways:

- Hybrid Testing is 25% faster than AFL to trigger bugs in lua and poppler
- Hybrid Testing finds bugs AFL does not and on average finds bugs as fast as AFL

Chapter 6

Future Work and Discussion

.

Throughout the completion of this work, one of the biggest drawbacks was the inability to fuzz many static analysis targets, detailed in Table 5.5. If Hybrid Testing is unable to fuzz a SA target, it cannot classify the result as a true or false positive. One potential solution to this problem is integrating additional directed fuzzers. We found that AFLGo and SieveFuzz were able to evaluate distinct applications; expanding the number of directed fuzzers could allow evaluation of new applications and code locations with Hybrid Testing.

We also found that Hybrid Testing is extremely sensitive to the performance of the underlying directed fuzzer. There is a significant difference in the results for SieveFuzz and AFLGo. SieveFuzz, when evaluating lua, triggered more of Magma's injected bugs than AFL and did so hours faster. AFLGo was consistently slower than AFL to detect the same Magma bugs. In the future, integrating additional directed fuzzers could help amortize the performance hit due to the underlying toolkit.

Another solution involves further refining static analysis targets before beginning fuzzing. Table 6.1 shows a breakdown of the categories of SA results for which Hybrid Testing was able to reach and trigger a bug. A key takeaway is that checkers associated with memory, including buffers and pointer manipulation, are a prime candidate for Hybrid Testing. Hybrid Testing could filter SA results by specific bug classes such as null pointer dereferences or buffer overflows to increase the chance a bug is detectable via fuzzing.

Another drawback to the current evaluation of Hybrid Testing is the limited fuzzing time. This work presents initial results using three runs of 24 hours of fuzzing. However,

Table 6.1: Static Analysis Target Categories Reached and Triggered by Hybrid Testing

Static Analyzer	Checker Name	Category	Reached	Triggered
CodeChecker	core.DivideZero	Logic error	Yes	No
CodeChecker	core.NullDereference	Logic Error	Yes	Yes
CodeChecker	misc-macro-parentheses	misc	Yes	Yes
CodeChecker	clang-diagnostic-unused-variable	clang	Yes	Yes
CodeChecker	cppcheck-nullPointerRedundantChecker	warning	Yes	No
FlawFinder	strcat	buffer	Yes	No
FlawFinder	getenv	buffer	Yes	Yes
FlawFinder	tmpfile	tmpfile	Yes	Yes
FlawFinder	fopen	misc	Yes	Yes
FlawFinder	char	buffer	Yes	Yes
FlawFinder	atoi	integer	Yes	Yes
FlawFinder	memcpy	buffer	Yes	Yes

many bugs within Magma, which are representative of real world CVEs, require more than 24 hours of fuzzing to be detected [36]. Further evaluation should attempt to run multi-day to multi-week long campaigns, collecting data from extended fuzzing and analyzing the impact of fuzzing time on results. Directed fuzzing is still an active area of research and though its performance is out of scope, extending fuzzing time could shed more light on Hybrid Testing's ability to better validate true and false positives.

In addition to longer fuzzing campaigns, parallel fuzzing could boost the performance of Hybrid Testing. Our evaluation utilizes Magma which restricts fuzzing to a single cpu core, disabling capabilities for parallel and distributed fuzzing. If instead of restricting fuzzing to a single core, mutiple cores were used, Hybrid Testing could realize better bug detection over time. However, many current fuzzing techniques (AFL included) also support parallel fuzzing, meaning that further quantitative evaluation is needed to determine parallel fuzzing's benefits.

A potential source of inaccuracy in our evaluation of Hybrid Testing is false positive classification. Currently, Hybrid Testing attempts to classify a bug as a false positive (meaning that it is a true negative) by directed fuzzing a static analysis target outside of Magma's injected bugs. If after 24 hours of fuzzing, Hybrid Testing does not find a bug at that location, we deem the static analysis target a false positive. This is potentially inaccurate if the bug requires more than 24 hours of fuzzing to be detected, does not cause a crash or memory bug detectable with address sanitization, or is unreachable through fuzzing. Potential solutions to this include increasing fuzzing time or manual inspection of a static analysis result, especially if the result is deemed high severity by a static analysis tool.

A final limitation of Hybrid Testing stems from its constitutent pieces: static analyzers and directed fuzzers. If a static analysis tool is inherently weaker at detecting a given class of vulnerabilities, Hybrid Testing will not have a static analysis target to begin analyzing. This could be mitigated by integrating additional static analyzers or fine tuning the settings used to run static analysis. If a directed fuzzer performs poorly when attempting to reach a target code location, Hybrid Testing will likely misclassify many bugs as false positives because the directed fuzzer fails to reach the bug location. Redundancy through additional directed fuzzers is again a solution to this problem, as well as amortization through running fuzzing campaigns multiple times.

6.1 Discussion

When we first performed the experiments evaluating Hybrid Testing's ability to triage false positives, we enabled Magma's Ideal Sanitizer. The Ideal Sanitizer caused every crash detected through fuzzing to also be detectable through ASan; it inserts a call to kill once a bug is triggered. This produced skewed results, often hundreds of thousands of crashing test cases per fuzzing run. When we examined the reports generated by ASan, we found that every report was due to Magma's Ideal Sanitizer. We did not find any of the bugs shown in Table 5.8. We disabled the Ideal Sanitizer and re-ran each experiment when fuzzing static analysis targets outside of the Magma injected bugs, which produced more coherent data which we were able to manually inspect.

We also acknowledge that directed fuzzers are severely impacted by their own ability to detect a bug. Both AFLGo and SieveFuzz rely on crashes to identify bugs, but not all bugs result in a crash. If a bug were to instead trigger a race condition but not cause a crash, it would go undetected by both AFLGo and SieveFuzz. While sanitizers can help mitigate these effects, sanitizers are computationally expensive and can greatly impact the performance of fuzzing. We hope that future research on directed fuzzing continues to improve bug detection capabilities without using crashes to verify bug existence.

Chapter 7

Conclusion

In this work we present Hybrid Testing, the first vulnerability detection technique with full codebase coverage and no false positives. Hybrid Testing is the next step in automated software testing, combining the strengths of static analysis and directed fuzzing. We detailed the design and implementation of Hybrid Testing and evaluated its accuracy across a corpus of open-source applications from the Magma fuzzing benchmark. We demonstrated that Hybrid Testing finds 17% more bugs than AFL when fuzzing Magma and can detect bugs up to 25% faster. It can be easily integrated into the development cycle, is highly generalizable to different tool kits of static analyzers and directed fuzzers, and scales well to large codebases. Future work should include integrating additional directed fuzzers and static analyzers, fuzzing Magma applications for more than 24 hours at a time, and applying Hybrid Testing to real world scenarios such as the applications from OSS-Fuzz. The initial results presented in this work show that Hybrid Testing is an improvement to traditional software testing techniques. Hybrid Testing promotes more secure software and provides software assurance through automatic bug detection and triage.

Appendix A

Tables

Table A.1: Standard Error Computed Across Three Fuzzing Runs for Each Magma Injected Bug — A standard error of zero indicates that only one sample was available while a '-' indicates that the bug was not reached or triggered.

BugID	Fuzzer	Reach Std. Err. (s)	Trigger Std. Err. (s)
LUA003	sievefuzz_inj_high	0	-
LUA003	sievefuzz_inj_medium	1.3	555
LUA004	afl	657.2	657.2
LUA004	sievefuzz_inj_high	169.5	169.5
LUA004	sievefuzz_inj_medium	12895.5	12895.5
PDF002	afl	2.4	-
PDF002	aflgo_inj_high	0	-
PDF002	aflgo_inj_medium	1.8	-
PDF003	afl	2.4	-
PDF003	aflgo_inj_high	2.4	-
PDF003	aflgo_inj_medium	0	-
PDF005	afl	2.4	-
PDF005	aflgo_inj_high	0	-
PDF005	aflgo_inj_medium	1.8	-
PDF007	afl	0	-
PDF007	aflgo_inj_high	2.4	-
PDF007	aflgo_inj_medium	1.6	-

Continued on next page

BugID	Fuzzer	Reach Std. Err. (s)	Trigger Std. Err. (s)
PDF008	afl	0	-
PDF009	afl	0	-
PDF009	aflgo_inj_high	2.4	-
PDF009	aflgo_inj_medium	2.2	-
PDF011	afl	0	22.7
PDF011	aflgo_inj_high	0	7487.6
PDF011	aflgo_inj_medium	0	22388.5
PDF012	afl	0	-
PDF012	aflgo_inj_high	0	-
PDF012	aflgo_inj_medium	0	-
PDF014	afl	0	-
PDF014	aflgo_inj_high	0	-
PDF014	aflgo_inj_medium	0	-
PDF016	afl	0	0
PDF016	aflgo_inj_high	0	9.4
PDF016	aflgo_inj_medium	0	221.4
PDF019	afl	0	-
PDF019	aflgo_inj_high	4.7	-
PDF019	aflgo_inj_medium	3.6	-
PDF021	afl	0	-
PDF021	aflgo_inj_high	2.4	-
PDF021	aflgo_inj_medium	1.6	-
PNG001	afl	0	-
PNG001	aflgo_inj_high	0	-
PNG001	aflgo_inj_medium	0	-
PNG003	afl	0	0
PNG003	aflgo_inj_high	0	0
PNG003	aflgo_inj_medium	0	0
PNG004	afl	0	-

Continued on next page

BugID	Fuzzer	Reach Std. Err. (s)	Trigger Std. Err. (s)
PNG004	aflgo_inj_high	0	-
PNG004	aflgo_inj_medium	0	-
PNG005	afl	0	-
PNG005	aflgo_inj_high	0	-
PNG005	aflgo_inj_medium	0	-
PNG006	afl	0	-
PNG006	aflgo_inj_high	0	-
PNG006	aflgo_inj_medium	0	-
PNG007	afl	0	0
PNG007	aflgo_inj_high	0	13415
PNG007	aflgo_inj_medium	0	9778.3
SND001	afl	4.7	-
SND001	aflgo_inj_high	0	-
SND001	aflgo_inj_medium	44.4	-
SND005	afl	0	2.4
SND005	aflgo_inj_high	0	844.2
SND005	aflgo_inj_medium	0	758.8
SND006	afl	4.7	-
SND006	aflgo_inj_high	0	-
SND006	aflgo_inj_medium	44.4	-
SND007	afl	4.7	-
SND007	aflgo_inj_high	0	-
SND007	aflgo_inj_medium	44.4	-
SND016	afl	4.7	-
SND016	aflgo_inj_high	0	-
SND016	aflgo_inj_medium	44.4	-
SND017	afl	4.7	1751.7
SND017	aflgo_inj_high	827.5	2307.4
SND017	aflgo_inj_medium	318.2	1602.7

Continued on next page

BugID	Fuzzer	Reach Std. Err. (s)	Trigger Std. Err. (s)
SND020	afl	2.4	-
SND020	aflgo_inj_high	825.9	-
SND020	aflgo_inj_medium	837.9	-
SND024	afl	4.7	-
SND024	aflgo_inj_high	0	-
SND024	aflgo_inj_medium	44.4	-
TIF002	aflgo_inj_high	18237.5	-
TIF002	aflgo_inj_medium	20892.6	-
TIF003	afl	0	-
TIF003	aflgo_inj_high	0	-
TIF003	aflgo_inj_medium	0	-
TIF006	aflgo_inj_high	18237.5	18237.5
TIF006	aflgo_inj_medium	20897	20897
TIF007	afl	7.1	1045
TIF007	aflgo_inj_high	39.6	14882.8
TIF007	aflgo_inj_medium	33.8	12409.9
TIF008	aflgo_inj_high	0	-
TIF009	afl	2915	2915
TIF009	aflgo_inj_high	5177	7369.6
TIF009	aflgo_inj_medium	16856.2	16848.3
TIF010	afl	2895.2	-
TIF010	aflgo_inj_high	9121.3	-
TIF010	aflgo_inj_medium	7607.2	-
TIF012	afl	0	3213.5
TIF012	aflgo_inj_high	0	4790.9
TIF012	aflgo_inj_medium	0	6876.8
TIF014	afl	7.1	3855.1
TIF014	aflgo_inj_high	39.6	7943.5
TIF014	aflgo_inj_medium	33.8	7942.5

Continued on next page

BugID	Fuzzer	Reach Std. Err. (s)	Trigger Std. Err. (s)
XML001	afl	0	-
XML001	aflgo_inj_medium	0	-
XML003	afl	0	-
XML003	aflgo_inj_medium	0	-
XML006	afl	0	-
XML006	aflgo_inj_medium	0	-
XML008	afl	1010.4	-
XML008	aflgo_inj_medium	1276	-
XML009	afl	0	-
XML009	aflgo_inj_medium	0	-
XML012	afl	0	-
XML012	aflgo_inj_medium	0	-
XML017	afl	0	21.2
XML017	aflgo_inj_medium	0	49.9

Bibliography

[1] American fuzzy lop (2.52b).

[2] Hamda Hasan AlBreiki and Qusay H Mahmoud. Evaluation of static analysis tools for software security. In *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98. IEEE, 2014.

[3] Aseel Alsaedi, Abeer Alhuzali, and Omaimah Bamasag. Effective and scalable black-box fuzzing approach for modern web applications. *Journal of King Saud University-Computer and Information Sciences*, 34(10):10068–10078, 2022.

[4] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, 2011.

[5] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Experiences using static analysis to find bugs. *IEEE Software*, 25:22–29, 2008. Special issue on software development tools, September/October (25:5).

[6] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[7] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[8] Domagoj Babic and Alan J Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th international conference on Software engineering*, pages 211–220, 2008.

[9] Al Bessey, Ken Block, Ben Chelf, Bryan Fulton Andy Chou, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] Hanmeet Kaur Brar and Puneet Jai Kaur. Differentiating integration testing and unit testing. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 796–798, 2015.

[12] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[13] Foteini Cheirdari and George Karabatis. Analyzing false positive source code vulnerabilities using static analysis tools. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4782–4788. IEEE, 2018.

[14] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[15] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. *arXiv preprint arXiv:2007.15943*, 2020.

[16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[17] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.

[18] Cve-2022-2602, October 2022.

[19] Openssl security advisory, November 2022.

[20] Cve-2022-4645, 2022.

[21] Heap buffer overflow in tiffcp.c:948, 2022.

[22] Aurelien Delaitre, Bertrand Stivalet, Paul Black, Vadim Okun, Terry Cohen, and Athos Ribeiro. Sate v report: Ten years of static analysis tool expositions, October 2018.

[23] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[24] Docker. Docker: Accelerated, containerized application development.

[25] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2440–2451, 2022.

[26] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Improving fuzzing assessment methods through the analysis of metrics and experimental conditions. *Computers & Security*, 124:102946, 2023.

[27] Ericsson. Codechecker.

[28] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[29] Flawfinder.

[30] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium*, pages 2577–2594, 2020.

[31] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.

[32] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[33] Google. Oss-fuzz - continuous fuzzing for open source software.

[34] NIST Software Quality Group. Nist samate.

[35] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A guided fuzzer to find buffer boundary violations. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., August 2013. USENIX Association.

[36] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[37] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50, 2008.

[38] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 230–243, 2021.

[39] Gerard Holzmann. Cobra — an interactive static code analyzer. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–1, 2017.

[40] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50, 2022.

[41] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[42] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.

[43] Wooseok Kang, Byoungho Son, and Kihong Heo. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1695–1708, 2022.

[44] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[45] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576. USENIX Association, August 2021.

[46] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 391–401. IEEE, 2019.

[47] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[48] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 249–259. IEEE Computer Society, 2019.

[49] Libfuzzer – a library for coverage-guided fuzz testing.¶.

[50] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *USENIX Security Symposium*, pages 1949–1966, 2019.

[51] Valentin JM 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*, 47(11):2312–2331, 2019.

[52] MITRE. Cwe - common weakness enumeration.

[53] Tukaram Muske and Uday P Khedker. Efficient elimination of false positives using static analysis. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 270–280. IEEE, 2015.

[54] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32, 2019.

[55] Muhammad Nadeem, Byron J Williams, and Edward B Allen. High false positive detection of security vulnerabilities: a case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 359–360, 2012.

[56] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[57] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for Use-After-Free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, San Sebastian, October 2020. USENIX Association.

[58] US Office of Personnel Management. Cybersecurity resource center, cybersecurity incidents.

[59] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2289–2306, 2020.

[60] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[61] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 181–198, 2020.

[62] Zachary P Reynolds, Abhinandan B Jayanth, Ugur Koc, Adam A Porter, Rajeev R Raje, and James H Hill. Identifying and documenting false positive patterns generated by static code analysis tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 55–61. IEEE, 2017.

[63] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[64] Nist software assurance reference dataset.

[65] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2595–2609, 2022.

[66] Divyanjali Sharma and Subodh Sharma. Thread-modular analysis of release-acquire concurrency. In *SAS*, 2021.

[67] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[68] Justin Smith, Lisa Nguyen Do, and Emerson Murphy-Hill. Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Proceedings of the Sixteenth Symposium on Usable Privacy and Security*, 2020.

[69] Source code analysis tools.

[70] 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 38th Annual Computer Security Applications Conference*, pages 388–399, 2022.

[71] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[72] Liang Sun, Wenfeng Lin, Shaoxian Shu, and Liuying Li. An interactive ranking algorithm for program static analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 242–249. IEEE, 2021.

[73] The Clang Team. Available checkers.

[74] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 413–430, Boston, MA, August 2022. USENIX Association.

[75] Kostyantyn Vorobyov and Padmanabhan Krishnan. Combining static analysis and constraint solving for automatic test case generation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 915–920, 2012.

[76] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *2009 International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE, 2009.

[77] 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 ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[78] What is the typical false positive rate of each coverity version?

[79] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. Fine-grained coverage-based fuzzing. *ACM Transactions on Software Engineering and Methodology*, 2023.

[80] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. Slf: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723. IEEE, 2019.

[81] Z3Prover. Z3prover/z3: The z3 theorem prover.

[82] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium*, volume 2022, 2022.

[83] 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*, 32(4):240–253, 2006.

[84] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[85] 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 *29th USENIX Security Symposium (USENIX Security 20)*, pages 2255–2269. USENIX Association, August 2020.