Compiler-assisted Adaptive Software Testing

Theofilos Petsios

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

Compiler-assisted Adaptive Software Testing

Theofilos Petsios

Modern software is becoming increasingly complex and is plagued with vulnerabilities that are constantly exploited by attackers. The vast numbers of bugs found in security-critical systems and the diversity of errors presented in commercial off-the-shelf software require effective, scalable testing frameworks. Unfortunately, the current testing ecosystem is heavily fragmented, with the majority of toolchains targeting limited classes of errors and applications without offering provably strong guarantees. With software codebases continuously becoming more diverse and complex, the large-scale deployment of monolithic, non-adaptive analysis engines is likely to increase the aforementioned fragmentation. Instead, modern software testing requires adaptive, hybrid techniques that target errors selectively.

This dissertation argues that adopting context-aware analyses will enable us to set the foundations for retargetable testing frameworks while further increasing the accuracy and extensibility of existing toolchains. To this end, we initially examine how compiler analyses can become context-aware, prioritizing certain errors over others of the same type. As a use case of our proposed approach, we extend a state-of-the-art compiler's integer error detection pipeline to suppress reports of benign errors by up to 89% in real-world workloads, while allowing for reporting of serious errors. Subsequently, we demonstrate how compiler-based instrumentation can be utilized by feedback-driven evolutionary fuzzers to provide multifaceted analyses targeting broader classes of bugs. In this direction, we present differential diversity (δ-diversity), we propose a generic methodology for offering state-aware guidance in feedback-driven frameworks, and we demonstrate how to retrofit state-of-the-art fuzzers to target broader classes of errors. We provide two such prototype implemen-

tations: Nezha, the first differential generic fuzzer capable of handling logic bugs, as well as SlowFuzz, the first generic fuzzer targeting complexity vulnerabilities. We applied both prototypes on production software, and demonstrate their effectiveness. We found that Nezha discovered hundreds of logic discrepancies across a wide variety of applications (SSL/TLS libraries, parsers, etc.), while SlowFuzz successfully generated inputs triggering slowdowns in complex, real-world software, including zip parsers, regular expression libraries, and hash table implementations.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would have not been made possible had it not been for all the amazing people that supported me throughout my PhD. I would first and foremost like to thank my advisor Angelos Keromytis, for giving me the opportunity to join the team at the Network Security Lab (NSL) of Columbia University, as well as my advisor Steven M. Bellovin, for his mentorship in the last stages of my studies. I am also grateful to all the faculty at Columbia with whom I had the privilege of collaborating, especially Suman Jana, Junfeng Yang and Roxana Geambasu.

I would also like to particularly thank Vasileios Kemerlis, who has been a mentor and friend throughout my studies, as well as all the bright colleagues with whom I had the privilege of discussing and collaborating. Particularly I would like to thank George Argyros, Vaggelis Atlidakis, Adrian Tang, Marios Pomonis, George Kontaxis, Michalis Polychronakis, Suphannee Sivakorn, Dimitris Mitropoulos, Kangkook Jee, Jason Polakis, and David Williams-King. I would not be who I am today as a researcher without you.

It goes without saying that I would like to wholeheartedly thank my family, who have supported me throughout my life with all means available to them, and still do. Finally, there is not enough words to express my gratitude to my wife, who has supported me more than anyone throughout my PhD and has been a companion and friend.

To Stella.

"..and in those days one often encountered the naive expectation that, once more powerful machines were available, programming would no crisis! How come?"

Dijkstra, "The Humble Programmer"

Chapter 1

Introduction

The impact of software errors is nowadays more dramatic than ever, with bugs resulting in human casualties [175, 102, 194, 182], catastrophic economic losses [1, 93, 80], massive outages [174, 127], even jeopardizing the national security of nations [193, 169]. As humanity relies more and more on software correctness, it is our duty as computer scientists to build robust error detection, prevention, and recovery tools.

1.1 Towards Software Correctness

Over the past decades, major research efforts have been made to mitigate the vast numbers of bugs found in production code. Such efforts involve, amongst others, abstract interpretation [43, 31], formal verification [92, 101, 12, 197, 72], static and dynamic analysis [190, 108, 126, 23], fuzzing [65, 206, 105], symbolic execution [21, 143, 28, 190, 90] as well as hybrid approaches that combine the above techniques [67, 65, 168, 70, 66, 34].

In addition to the advances in the areas of testing and verification, significant progress has also been made in the fields of programming languages and compilers [178, 107, 100]: existing programming languages are continuously improving, whereas languages with increasingly stronger memory [55, 113], type-safety [58], and

correctness guarantees [172] are being developed and deployed in production. Likewise, compilers are constantly becoming more robust and incorporate passes to proactively guard against programming errors [151, 154, 107, 100, 155, 167].

Unfortunately, despite the aforementioned advances, the current software ecosystem is far from sound. This is partly due to the fact that all existing binary application testing techniques suffer from limitations. For instance, static analysis tools predominantly suffer from false positives and inter-procedural analysis constraints [98], dynamic techniques are slow [133] and unsound [53], whereas approaches based on symbolic execution do not scale and suffer from path explosion [143]. To make matters worse, modern binary application testing frameworks do not equally address all types of errors. Instead, in their majority they target few classes of bugs, such as those related with temporal or spatial safety [4], whereas tools that attempt detection of different families of errors such as logic bugs, race conditions, complexity, data-leakage or side-channel vulnerabilities, are scarce. Finally, not only are testing tools limited in their analysis or scope, but any non-trivial property of a Turing-complete programmming language is undecidable [39, 153, 26, 207, 51]. As a consequence, a multitude of bugs are making their way into production code year after year [185], with even well understood errors like buffer overflows plaguing commercial off-the-shelf software [185, 52].

Except for the limitations of the analysis and scope of current testing frameworks, the software ecosystem status quo is largely shaped by the existence of massive legacy codebases, which are critical to the operation of currently deployed infrastructure. The necessity for such backwards compatible software dictates that, despite the fact that new programming languages with stronger properties are constantly being developed, their large-scale adoption is impeded due to cost, usability, lack of expertise, or performance constraints [117, 146]. Moreover, it is often the case that these exact new technologies are built using non memory-safe languages,

predominantly C and C++, which are still widely deployed and serve as the building-blocks for performance-critical applications, operating systems or other programming languages. Such complex dependencies are not without cost: although several of the foundational codebases (e.g., `libc`) have been thoroughly tested throughout the years, software errors might be lurking at all levels of the development stack, or in the hardware-software interaction.

From all the above it is made clear that the effort towards improving software quality is twofold: on one hand we need better programming languages, compilers, software development practices and frameworks, and on the other we need to improve and extend the existing testing infrastructure, both to address errors in new codebases but, more importantly, as a necessity in order to continue improving legacy software. This thesis focuses on improving the state-of-the-art in the current binary application testing ecosystem and presents novel techniques by which the modern toolchains can be retrofitted to provide finer-grained analyses and target broader classes of errors. In the rest of this Chapter, we elaborate on the limitations of the current infrastructure that motivated this work, as well as outline the contributions this thesis makes towards overcoming the aforementioned inefficiencies.

1.2 Adaptive Testing for the Modern Era

Although previous work has set the foundations for exploring the state space of large applications, the testing ecosystem has traditionally followed a *monolithic* approach towards bug detection, in the sense that errors are reported in a quantitative rather than a qualitative manner. For instance, compilers report violations of language specifications while being predominantly agnostic to the context in which these violations appear. However, not all errors are equally critical: some might be exploitable whilst others not, some may result in resource underutilization, whereas others might

constitute developer-intended violations of the specification. Of course, compilers are not the only frameworks adopting such a context-agnostic design: fuzzers and symbolic execution engines utilize different metrics to explore the application space, and do so traditionally without any notion of selectivity regarding *which* application components might be more relevant to the performed analysis.

Another popular characteristic of modern testing frameworks is that, predominantly, they focus on particular bug classes, frequently even targeting only specific programs. For instance, MongoDB's Javascript fuzzer [69] only targets the MongoDB engine and builds inputs by maintaining domain-specific valid Abstract Syntax Trees, tailored to the MongoDB API. Such tailor-made targeting is not surprising given the difficulty of the task in hand: different systems have different characteristics and break in different ways, thus it is natural that one-fits-all testing tools are doomed to be incomplete, especially when solutions that eradicate the existence of even one specific bug class are rare and hard to design. However, a downside of this approach is that, due to the fact that the analyses performed are usually very specific to the particular application being tested, reusing parts of a framework in different applications is impeded. Likewise, little or no knowledge can be transferred from one framework to another, or across different testing sessions of the same application. As a result, a fuzzer that is very successful in testing a particular TLS library might not be equally successful at testing other TLS libraries, and even more so when targeting completely different applications such as media players. Finally, it is usually the case that, even if a testing tool is application-agnostic, it performs its testing in a stateless manner, with previous testing sessions not affecting future invocations.

Characteristics such as the ones outlined above (e.g., non-transferability of knowledge as well as adopting monolithic, stateless and non-adaptive engines), have led to a *fragmented testing ecosystem*, where particular tools are useful only for limited sets of applications and bug types, without offering (provably) strong guarantees. With

software applications becoming increasingly diverse and complex, it will be harder and harder for the testing infrastructure to keep-up with change without further increasing this fragmentation. The hypothesis behind this thesis is that *adopting context-aware analyses increases the accuracy and extensibility of existing toolchains*. To better understand this motivation, let us elaborate on some differences between the proposed approach and current monolithic, non-adaptive designs.

For one, adopting a monolithic design such as the ones described previously may hinder testing workflows, since analysts, in order to discern which errors are more relevant to their analysis, are usually required to perform manual filtering of the generated error reports, a process which is non-trivial for large codebases. Such filtering, however, can be performed automatically by the testing framework, if the latter is able to support *prioritization* of certain analysis states over others. In such cases, the analysis essentially becomes *context-aware*, since each state may be given an attribute based on its relevance to particular characteristics of interest (see Section 3.2). This, for instance, can be achieved by assigning scores to the different states of the analysis engine based on a context-indicative metric. Thus, a tool that reports integer errors ignoring the context in which they occur could be augmented to prioritize analysis of errors that involve inputs controlled by the user, or errors that propagate into memory allocations. Likewise, a symbolic execution framework that attempts to blindly solve constraints to explore an application's state could prioritize solutions to constraints involving input sanitization.

Contrary to monolithic, context-agnostic architectures, context-aware designs offer the additional benefit of providing a basis upon which to construct *adaptive* analyses: if the analysis engine is capable of maintaining an internal hierarchy of its states, it may adjust this hierarchy either at runtime or at bootstrapping to target different areas of the program under scrutiny or different classes of errors. Thus, a fuzzer that explores new components of an application to maximize code coverage,

may adapt its engine to maximize the number of encountered memory deallocations instead. Moreover, being context-aware increases the selectivity of the performed analyses and therefore, by limiting their search space, their accuracy. This property is beneficial towards overcoming common limitations linked with state explosion, as well as enables the analysts to fine-tune their testing, focusing on specific areas of interest in the application under scrutiny. Finally, it is important to note that by augmenting existing toolchains to support adaptive analyses, essentially by adding the appropriate abstractions into the analysis and instrumentation engines, current frameworks are still capable of reusing the components of their infrastructure that are truly context-agnostic, without any modifications, to target broader classes of bugs. For instance, an evolutionary fuzzing framework will be able to re-use its components that handle the input corpus evolution, however steering its input generation towards different types of bugs like crashes, complexity vulnerabilities or logic errors, by simply adapting its context-aware logic to the different use-cases. Thus, by adopting such abstractions to separate context-aware and context-agnostic components, the scope of existing tools can increase to include additional bug classes while simultaneously achieving an increase in the frameworks' *modularity*, which in turn will enable better integration with other toolchains.

1.3 Compiler-assisted Adaptive Software Testing

Prior to elaborating on how modern systems can provide more adaptive and context-aware testing, it is necessary to understand the core structural components of their analysis engines. Over the past years, increasingly more frameworks have adopted white-box and grey-box architectures that make use of the internal state of the application being tested. This internal state provides feedback to the testing engine and its use yields superior results compared to black-box approaches, since more infor-

mation is available to the analysis. As a result, such designs find many applications in fuzzing and concolic execution frameworks, where the main engine performs some form of bookkeeping, for instance through the number of total Control Flow Graph (CFG) edges accessed, or the number of new sets of solutions to a given constraint solving problem. This bookkeeping, which is performed during the execution of a testing session, subsequently affects future testing invocations.

In order to enable these feedback-driven analyses, the application is usually instrumented either using dynamic binary instrumentation (DBI), directly through the compiler toolchain, or executes under a hypervisor, and the gathered information from each session is then made available to the testing framework. Dynamic techniques have the advantage of not requiring access to the application source code and offer large versatility with respect to the available instrumentation choices, however are slower than static methods, and are often cumbersome to use by the vast majority of developers. On the other hand, compiler-based instrumentations are readily-available and offer fine-grained tracking, but are less flexible than dynamic approaches due to the fact that any analysis is performed once, at the time of compilation. However, empirical evidence suggests that developers are not widely adopting a software testing or hardening scheme, unless the latter is simultaneously effective and usable [50, 77]. It is thus of no surprise that analyses that are integrated into compiler toolchains [154, 82] are amongst the most popular due to their ease of use, effectiveness, and performance.

As mentioned in the previous Section, although compiler-based tools detect a variety of bugs ranging from undefined behavior to integer errors, invalid memory accesses, memory leaks, etc., they are providing *generic* protections, in the sense that they are not context-aware, and thus do not adapt their logic to particularly match the state of the application been tested. This is sub-optimal for the prospective analyst, as bugs that are more critical in the context of a particular application are not

given a higher priority based on their severity. In the first part of this thesis, we will outline a methodology under which such information can be provided by the compiler toolchain. In particular, by adding specially crafted static analysis passes and combining them with compiler-constructed dynamic monitors, we will examine how compilers' analyses can become *context-aware*, prioritizing certain errors of the same type over others. Subsequently, we will demonstrate how compiler-based instrumentation can be utilized by feedback-driven evolutionary fuzzers to provide multifaceted analyses targeting broader classes of errors. To this end, we will extend a state-of-the-art fuzzer to target, except for crash-inducing bugs, logic errors, as well as complexity vulnerabilities. We will prototype our designs in Nezha, the first differential generic fuzzer capable of finding logic bugs, as well as SlowFuzz, the first generic fuzzer targeting complexity vulnerabilities. Note that both Nezha and SlowFuzz have common core components, and can also target traditional crash-inducing vulnerabilities. However, being adaptive, they are able to target their analysis engine on different types of bugs as needed. As noted previously, we believe that such modular designs are essential towards achieving *generally applicable* testing: in the concluding Chapter of this thesis, we will discuss how the contributions made in this work can be utilized towards this end, as well as outline promising future directions for research on the topic.

1.4 Contributions

In short, this thesis makes the following contributions:

- We present a methodology to augment monolithic frameworks to support context-aware testing.
- We introduce *differential diversity* (δ-diversity), a novel metric to be used for selective input generation guidance. We demonstrate δ-diversity can be utilized

to achieve testing in a white-box or grey-box manner, and also demonstrate how δ-diversity generalizes known black-box schemes.

- We experimentally evaluate δ-diversity against coverage-based guidance schemes and demonstrate that it yields superior results in the context of differential testing while simultaneously achieving equally good performance in single-application testing, with respect to coverage and bugs found.
- We design, implement and evaluate the first fuzzing framework particularly targeted towards differential testing.
- We design, implement and evaluate the first resource-feedback driven fuzzing engine targeting complexity vulnerabilities.
- We report multiple previously unknown security vulnerabilities in security-critical applications and make all the presented testing frameworks publicly available for the community.

1.5 Dissertation Roadmap

The rest of this work is outlined as follows: Chapter 2 provides background information on modern application testing and analyzes the core state-of-the-art techniques relevant to this thesis. Chapter 3 discusses compiler toolchains in the context of testing and how their analyses can be augmented to achieve context-aware error reporting, whereas Chapters 4 & 5 examine how compiler-based instrumentation can be used in modern evolutionary fuzzing frameworks to provide adaptive testing. To this end, Chapter 4 introduces differential diversity (δ-diversity), a novel methodology for peforming differential testing, and outlines NEZHA, the first, to the best of our knowledge, differestial fuzzer targeting logic bugs. Chapter 5 outlines how the core components of an evolutionary fuzzer such as the one used in NEZHA can be utilized

to target different types of bugs. The methodologies presented are prototyped and evaluated in SlowFuzz, which shares the same core components as Nezha, however adapts its analysis engine to target complexity-vulnerabilities instead. Finally, Chapter 6 summarizes this work and discusses points not addressed in this thesis, as well as promising directions for future research.

Chapter 2

Related Work

In this Chapter, we outline some lines of work which are closer to the material discussed in this thesis. Although software testing covers many different areas of research that can often be orthogonally combined with the techniques presented in this thesis, for our purposes we will not discuss a series of topics that are beyond the scope of this work, such as formal verification [92, 101, 12, 197, 72], abstract interpretation [43, 31] or model checking [121, 60].

2.1 Static Analyses & Compiler Toolchains

A series of compiler-assisted tools target spatial and temporal safety errors, and attempt to prevent against their exploitation. These tools can be separated into three main categories, based on whether they provide protection using pointer-based, object-based or tripwire approaches.

Object-based techniques: J&K [87] improved upon techniques using fat pointers by utilizing a splay tree that holds a bounds table for created objects, and checking both pointer arithmetic as well as `load`/`store`, operations however is limited by pointers in external libraries and suffers from program intermediate pointer use. CRED [149] creates an out-of-bounds (OOB) in the heap for every stored OOB value, whereas

Dhurjati et al. improve upon CRED by utilizing automatic pool allocation and maintaining a different splay tree per pool. Finally, Baggy Bounds Checking (BBC) [**bbc**] uses a buddy allocator to allocate memory at the granularity of fixed slots, and store the respective bounds in a bounds array [4].

Pointer-based techniques: SoftBound [123] provides spatial safety by associating base and bounds info with each pointer in a disjoint-data space, and checking bounds upon dereferences, propagating the respective metadata upon function calls. CETS [122], on the other hand, provides temporal safety for C programs by maintaining a unique allocation key and lock address for each pointer and changing the lock every time the respective memory region is deallocated. Y&H [204] perform static analysis to identify unsafe pointers (that might go out of bounds), as well as their points-to sets, and tag each byte of memory as appropriate or not (maintaining a shadow mirror of memory), keeping a runtime set of referents for each pointer. Finally, MemSafe [160] provides both spatial and temporal safety guarantees by modeling temporal errors as spatial errors and keeping a hybrid metadata representation for objects and pointers.

Tripwire approaches: In the past years, several testing schemes utilizing red-zone guarding of memory (insertion of memory-protected memory regions around allocated memory) have been integrated into compiler toolchains, to detect out-of-bounds accesses [154]) use-after free vulnerabilities and data races [82]. Such tools advance compiler-assisted dynamic detection and do not generate false positives, however come at a high performance cost and have no notion of selectivity in case a bug is found.

Static analysis tools provide more sophisticated analyses and achieve larger code coverage but generally suffer from a high rate of false positives. For instance, KINT [190] is a static tool that generates constraints representing the conditions under which an integer overflow may occur. It operates on LLVM IR and defines untrusted

sources and sensitive sinks via user annotations. KINT avoids path explosion by performing constraint solving at the function level and by statically feeding the generated constraints into a solver. After this stage, a single path constraint for all integer operations is generated. Unfortunately, despite the optimization induced by the aforesaid technique, the tool's false positives remain high and there is a need for flagging false positives with manual annotations in order to suppress them. Moreover, KINT attempts to denote all integer errors in a program and does not make a clean distinction between classic errors and errors that constitute vulnerabilities.

Finally, a family of static analysis tools that are relevant to the discussion presented in Chapter 3 are those particularly targeted at detection of integer errors: IntPatch [209] is built on top of LLVM [97] and detects vulnerabilities utilizing the type inference of LLVM IR. IntPatch uses forward & backward analysis to classify sources and sinks as sensitive or benign. SIFT [106] uses static analysis to generate input filters against integer overflows. If an input passes through such filter, it is guaranteed not to generate an overflow. Initially, the tool creates a set of critical expressions from each memory allocation and block copy site. These expressions contain information on the size of blocks being copied or allocated, and are propagated backwards against the control flow, generating a symbolic condition that captures all the points involved with the evaluation of each expression. The free variables in the generated symbolic conditions represent the values of the input fields and are compared against the tool's input filters. IntScope [189] decompiles binary programs into IR and then checks lazily for harmful integer overflow points. To deal with false positives, IntScope relies on a dynamic vulnerability test case generation tool to generate test cases which are likely to cause integer overflows. If no test case generates such error, the respective code fragment is flagged appropriately. Finally, RICH [23] is a compiler extension which enables programs to monitor their execution and detect potential attacks exploiting integer vulnerabilities. Although RICH is very lightweight,

it does not handle cases of pointer aliasing and produces false positives in cases where developers intentionally abuse the undefined behavior of C/C++ standards.

2.2 Unguided Testing

Unguided testing tools generate test inputs independently across iterations without considering the test program's behavior on past inputs. *Domain-specific* evolutionary unguided testing tools have successfully uncovered numerous bugs across a diverse set of applications [118, 79, 148, 88, 84]. Another parallel line of work explores building different *grammar-based* testing tools that rely on a context free grammar for generating test inputs[114, 110]. LangFuzz [74] uses a grammar to randomly generate valid JavaScript code fragments and test JavaScript VMs. TestEra [111] uses specifications to automatically generate test inputs for Java programs whilst *lava* [161] is a domain-specific language designed for specifying grammars that can be used to generate test inputs for testing Java VMs.

2.3 Guided Testing

Evolutionary testing was designed to make the input generation process more efficient by taking program behavior information for past inputs into account, while generating new inputs [131]. Researchers have since explored different forms of code coverage heuristics (e.g., basic block, function, edge, or branch coverage) to efficiently guide the search for bug-inducing inputs. Coverage-based tools such as AFL [206], libFuzzer [105], and the CERT Basic Fuzzing Framework (BFF) [75] refine their input corpus by maximizing the code coverage with every new input added to the corpus. Another line of research builds on the observation that the problem of new input generation from existing inputs can be modeled as a stochastic process. These tools leverage a diverse set of statistical techniques to drive input generation [99, 38, 20].

Chen et al.'s perform differential testing of JVMs using MCMC sampling for input generation [38] however their approach is domain-specific (i.e., requires details knowledge of the Java class files and uses custom domain-specific mutations). Likewise, Veggalam et al. [183] combine genetic programming with grammar-based fuzzing to test Javascript interpreters. Vuzzer [145] utilizes control- and data-flow analysis, to prioritize deep paths when mutating inputs, as well as determine where and how to mutate those inputs.

Finally, a series of techniques for similarity code detection [200], function identification [159], and automata learning [7, 162], as well as synthesis from input/output examples of SQL queries [186] or input grammars [10], can be combined with guided testing methodologies such as the ones described above.

2.4 Differential Testing

Differential testing [116] shares parallels with N-version programming [36]. Both aim to improve the reliability of systems by using independent implementations of functionally equivalent programs, provided that the failures (or bugs) of the multiple versions are statistically independent. Researchers have leveraged this approach to find bugs across many types of programs, such as web applications [35], different Java Virtual Machine (JVM) implementations [38], various security implementations of security policies for APIs [165], compilers [202] and multiple implementations of network protocols [24]. KLEE [28] used symbolic execution to perform differential testing, however suffers from scalability issues. SFADiff [7] performs black-box differential testing using Symbolic Finite Automata (SFA) learning, however, can only be applied to applications such as XSS filters that can be modeled by an SFA. Brubaker et al.'s unguided differential testing system that synthesizes *frankencerts* by randomly combining parts of real certificates [22]. They use these syntactically valid certificates

to test for semantic violations of SSL/TLS certificate validation across multiple implementations. Chen et al. build on top of frankencerts to perform coverage-guided differential testing of SSL/TLS implementations using MCMC sampling in Mucerts [37]. However, Mucerts requires knowledge of the partial grammar of the X.509 certificate format and its input generation is very slow, requiring multiple days to generate even 10,000 inputs. Finally, besides testing software, researchers have applied differential testing to uncover program deviations that could lead to malicious evasion attacks on security-sensitive programs. Jana et al. use differential testing (with manually crafted inputs) to look for discrepancies in file processing across multiple antivirus scanners [83]. Recent works have applied differential testing to search for inputs that can evade machine learning classifiers for malware detection [199, 96].

2.5 Symbolic Execution

Symbolic execution [90] is a white-box technique that executes a program *symbolically*, computes constraints along different paths, and uses a constraint solver to generate inputs that satisfy the collected constraints along each path. KLEE [28] uses symbolic execution to generate tests that achieve high coverage for several popular UNIX applications, however, due to path explosion, it does not scale to large applications. UC-KLEE [90, 144] aims to tackle KLEE's scalability issues by performing under-constrained symbolic execution, i.e., directly executing a function by skipping the whole invocation path up to that function. However, this may result in an increase in the number of false positives. To mitigate path explosion, several lines of work utilize symbolic execution only in certain parts of their analysis to aid the testing process, and combine it with concrete inputs [29]. Another approach towards addressing the limitations of pure symbolic execution is to outsource part of the computation away from the symbolic execution engine using fuzzing [67, 65, 168,

70, 66, 34].

Chapter 3

Compiler-assisted Testing

3.1 Background

Before examining how compiler analyses can be augmented to provide context-aware error reporting, let us outline some of the ways in which they are used to prevent the occurrence and exploitation of software errors. Subsequently, we will provide a methodology to augment state-of-the-art toolchains to achieve adaptive testing.

By construction, compilers and interpreters check the syntactic correctness of the code they analyze and report deviations from the respective language specifications. This is usually achieved during the different passes of the compilation toolchain [3]. Additionally to reporting syntactic errors and relevant warnings, however, over the past decades compilers have also been actively used in a multitude of different contexts, including defending against stack smashing [44, 40, 133] and Return Oriented Programming (ROP) [130, 141, 103] exploits, as well as to provide stronger code, data pointer [95, 25, 5] control flow [177, 112, 89] and data flow [32] integrity properties. This is usually achieved by re-organizing the code of the application appropriately at compile-time, as well as by inserting callbacks, dynamic monitors, instrumentation hooks and red-zones (memory-protected regions) at the binary. The aforementioned

techniques have found extensive use not only in binary hardening defenses, but also in compiler-assisted testing frameworks. Most notably, the combination of red-zone insertion with the use of dynamic monitors has been used to detect memory safety errors, such as invalid memory accesses or use-after-free violations [154, 181, 82], whereas instrumentation hooks and callbacks that provide execution information [151] are extensively used by fuzzers or other feedback-driver testing frameworks [206, 105, 134, 135, 20, 168].

Typically, compiler-assisted testing toolchains can be separated into two main categories, based on whether testing is performed statically or dynamically. In cases were testing is performed purely statically, several analysis passes that may or may not be receiving external inputs are performed, and error reports are generated without the need to execute the application. In dynamic compiler-assisted testing, on the other hand, the binary is appropriately modified at compile-time and subsequently executed, and violations are reported at runtime. For instance, AddressSanitizer [154] inserts red-zones around all allocated memory, and, during program execution, monitors accesses in these red zones, reporting the respective out-of-bounds error. Similarly, modern evolutionary fuzzers exercise different inputs on the application being tested and, after each input execution, examine how many *previously unseen* basic block edges have been accessed during that execution: if the respective input exercised edges that were not encountered before, it is added into the input corpus to affect the generation of future inputs. The information on what edges were accessed is made available to the fuzzer via the appropriate hooks that the compiler inserted at each basic block during compilation. Likewise, support for the bookkeeping of the accessed edges is also provided by the compiler, and the appropriate callbacks to manipulate this bookkeeping metadata is exposed to the fuzzing engine.

Regardless of whether errors are reported with or without executing inputs, however, as mentioned in Chapter 1, modern compiler-based techniques predominantly

report errors indiscriminately rather than qualitatively. Section 3.2 describes context-aware testing and how it can be used to provide such qualitative information, whereas Section 3.3 outlines a methodology under which compiler analyses can be augmented to provide qualitative information, and presents the design of one such prototype implementation focusing on context-aware reporting of integer errors.

3.2 Augmenting Compiler Analyses

Static analyses are known to suffer from false positives and state-explosion limitations [98, 53]. However, as mentioned in Chapter 1, context-aware designs offer more targeted analyses, limiting the search space for the analysis engine. In the context of this work, we classify an analysis as context-aware [1] if its results can be grouped into *more than one* (possibly overlapping) sets, each of which is semantically associated with a single user-defined property. If such grouping is feasible, then the respective resulting subsets can be prioritized by the analysis engine, with their ordering denoting a hierarchy in the (context-dependent) semantic characteristics of each group's properties.

For instance, suppose we want to modify a coverage-based fuzzer so that it focuses its testing into the application's source code rather than in external libraries that may be linked into the binary. Normally, evolutionary coverage-based fuzzers keep track of the unique basic block edges of the application that have been encountered so far, and favor inputs that exercise new edges [206, 105]. However, simply partitioning edges into two groups based on whether they have been accessed in previous sessions does not suffice, in itself, to provide context-aware guidance, since no semantic property is associated with both sub-groups: for each input that executes, the fuzzer knows that it accessed edges that may or may not have been encountered previously, but all

[1] The term should not be confused with testing of context-aware applications [192, 150, 9, 205].

edges that are discovered fall within the same bucket and are treated equally, exactly as compilers treat, for instance, all integer errors equally. Instead, one alternative (naive) way in which the fuzzing engine could provide context-aware guidance would be to split the covered edges into two sets, based on whether they belong to external libraries (e.g., `libc`) or the source code of the application and subsequently, when forming the input corpus, favor inputs that contributed more edges to the group of interest, essentially "steering" the testing towards exploring the respective regions.

At this point we should mention that, for the above example, even if a different metric that is more expressive of the application state is used instead of code coverage, unless it is tied semantically with particular attributes of interest (either user-defined or automatically determined), and used to prioritize certain states over others, it cannot be used for context-aware guidance as described in this work. To make this point clearer, let us consider that instead of using code coverage, our fuzzer instead bases its input generation on the different call stack traces from the execution of each input. Thus, every time an input executes, the respective function call sequences are recorded (given the appropriate compiler support or using dynamic binary instrumentation/binary rewriting), and, if an input exercises a call stack trace that was not encountered before, it is preserved in the input corpus. Despite the fact that each call frame stack is representative of the state of the program, no semantic information is tied with that particular state, thus the fuzzer will not be able to prioritize certain states over others, and instead is forced to blindly explore the application space. Instead, if there is a grouping of the possible call stacks, e.g., depending on whether they involve particular addresses/functions of interest (such as functions performing cryptographic operations, parsing, etc.) then, using that grouping it will be possible to prioritize the generated inputs, and this prioritization will reflect the significance of the respective semantic groups.

As we will demonstrate in the rest of this thesis, this methodology can provide

fine-grained testing without impacting performance or the total errors reported, and can be equally applied to both static and dynamic techniques.

3.3 Context-aware Compiler Analyses

As mentioned in Section 3.1, compiler-based toolchains may locate errors either statically or dynamically. Thus, context-aware clustering can be achieved either during the compilation phase or at runtime, using inline dynamic monitors that rely on the compiler's instrumentation. In this Section we will provide an example of a real-world implementation of the approach proposed in Section 3.2, modifying a state-of-the-art compiler, so that the latter may provide context-aware discovery of integer errors, in which bugs are prioritized based on their likelihood to be exploitable or to have unintended consequences in software execution.

3.3.1 Motivation

Error debugging is amongst the biggest development hurdles [33, 61, 11], since error reports for production applications may be prohibitively large, with several errors being particularly hard to debug [203]. Requiring human analysts to manually examine errors in large projects is costly and, in certain software deployment scenarios, even infeasible. Ideally, developers should be receiving reports with low or zero false positives, and should be able to prioritize which errors they should debug first, based on their criticality. Unfortunately, static analysis frameworks and compiler toolchains generate reports without any prioritization of certain errors over others, and thus it is often infeasible to discern which errors are more critical directly from the generated report and without additional analysis. This problem becomes even worse in cases where the software code is conflicting with compilers' optimizations, or in cases involving undefined behavior. In these scenarios, developers may intentionally be

utilizing exotic or erroneous code constructs to achieve custom functionality or enhanced performance. However, such constructs are detected by compilers and linters as errors, and are reported in bulk together with other errors that may be lurking in the code and which may be completely unknown to the software authors. In the following, we will examine how compiler passes can be augmented to aid developers in such scenarios. Particularly, we will augment the LLVM compiler toolchain [97] to achieve context-aware reporting of integer errors, which are frequently encountered in contexts involving undefined behavior and optimizations.

3.3.2 Use-case: Context-aware Integer Error Reporting

3.3.2.1 Integer Errors and Undefined Behavior

Although the C and C++ language standards explicitly define the outcome of most integer operations, a number of corner cases are left undefined. As an example, the C11 standard considers an `unsigned` integer overflow as a well-defined operation, whose result is the minimum value obtained after the wrap-around, while leaving `signed` integer overflows undefined. This choice facilitates compiler implementations to produce optimized binaries [191]. For instance, signed integer overflows (or underflows) enable compiler developers to implement an optimization that infers invariants from expressions such as $i+1 > i$ and replaces them with a constant Boolean value [3].

Table 3.1 lists special cases of integer operations and their definedness. It should be noted that although more instances of undefined behavior (not necessarily restricted to integer operations) are declared in the language specification, we only consider integer operations for this work.

As in practice not all cases of undefined behavior necessarily result in actual errors, the difficulty of dealing with these types of bugs lies in distinguishing *c*ritical integer errors from intended violations of the standard. The intention of a developer, however, cannot be formally defined or automatically derived, as the code patterns

Table 3.1: Examples of defined and undefined arithmetic operations according to the C/C++ language specification.

Arithmetic Operation	**Definedness**
Unsigned overflow (underflow)	defined
Singed overflow (underflow)	undefined
Signedness conversion	undefined*
Implicit type conversion	undefined*
Oversized/negative shift	undefined
Division by zero	undefined

*if value cannot be represented by the new type

used in a piece of code are deeply related to the author's knowledge, preference, and programming style. Although writing code that intentionally relies on undefined operations is generally considered a bad programming practice (as the outcome of those operations can be arbitrary, depending on the architecture and the compiler), there are several cases in which the community has reached consensus on what is the expected behavior of the compiler in terms of the generated code, mainly due to empirical evidence. This explains why idioms that take advantage of undefined behavior are still so prevalent: although according to the standard the result of an operation is undefined, developers have an empirically derived expectation that compilers will always handle such cases in a consistent manner. This expectation creates serious complications whenever developers check the validity of their code with state-of-the-art static analysis tools. These tools evaluate code based on strict conformance to the language specification, and consequently generate a large amount of false positives. As a result, the generated reports are often overlooked by developers who struggle to spot which of the reported bugs are actual errors. Unfortunately, tools based on dynamic code analysis also do not provide strong guarantees in these cases, as they suffer from low code coverage.

Thus, while the task of automatically detecting undefined arithmetic operations is relatively easy, the true difficulty lies in identifying critical or unintended violations

the language standard. To further illustrate the complexity of this issue, let us consider the code pattern presented in Figure 3.1 which one of the most prevalent cases of undefined operation abuse.

```
1   UINT_MAX = (unsigned) -1;
2   INT_MAX = 1 << (INT_WIDTH - 1) - 1;
```

Figure 3.1: Widely used idioms that according to the standard correspond to undefined behavior.

The two C statements of Figure 3.1 are often intentionally used by developers mainly to achieve persistent representation across different system architectures. Both are based on assumptions on the numerical representation used by the underlying system (two's complement). Line 1 shows a case of signedness casting in which the original value cannot be represented by the new type. In Line 2, a shift operation of `INT_WIDTH - 1` is undefined[2] but it conventionally returns the minimum value of the type, while the subtraction operation incurs a signed underflow which is also undefined. Although these cases are violations of the language standard, the desirable operation of an integer overflow checker would be to not report them as of high risk, as they most likely correspond to developer-intended violations, and from developers' view, in case they were indeed intentional, would be considered false positives [53].

On the contrary, in the example of Figure 3.2, the unsigned integer variable (`alloc_size`) might overflow as a result of the multiplication operation at line 5. This behavior is well-defined by the standard, but the overflow may result in the allocation of a memory chunk of invalid (smaller) size, and consequently, to a heap overflow. An effective arithmetic error checker should be able to identify such potentially exploitable vulnerabilities and give high priority to the respective error, as most likely the developer did not intend for this behavior.

[2]According to the C99 and C11 standards. The C89 and C90 (ANSI C) standards define this behavior.

```
1  /* struct containing image data, 10KB each */
2  img_t *table_ptr;
3  unsigned int num_imgs = get_num_imgs();
4  ...
5  unsigned int alloc_size = sizeof(img_t) * num_imgs;
6  ...
7  table_ptr = (img_t*) malloc(alloc_size);
8  ...
9  for (i = 0; i < num_imgs; i++)
10   { table_ptr[i] = read_img(i); }  /* heap overflow */
```

Figure 3.2: An unsigned integer overflow as a result of a multiplication (line 5), which results in an invalid memory allocation (line 7) and unintended access to the heap (line 10).

3.3.2.2 Design

In order to achieve the integer error reporting at granularity described previously, we build upon the following observations:

- Integer errors may be developer-intended if they fall within known programming constructs as those of Figure 3.1. From the perspective of developers, the respective errors are generally considered of low-priority.

- Integer errors that originate from trusted sources that are not user-controlled and do not affect sensitive operations (memory-allocations, system persistent data, permissions, etc.) are not likely to be exploitable. Such errors, for instance, are integer errors that originate from constant assignments, or errors whose sources are non-security critical system calls and trusted library functions (e.g., `uname()`).

- Errors that may propagate into sensitive operations such as system calls, memory allocations, etc., or errors that originate from untrusted sources should receive high priority.

Based on the above observations, we can create a semantic partitioning of the different errors reported by the compiler and let its engine appropriately prioritize them

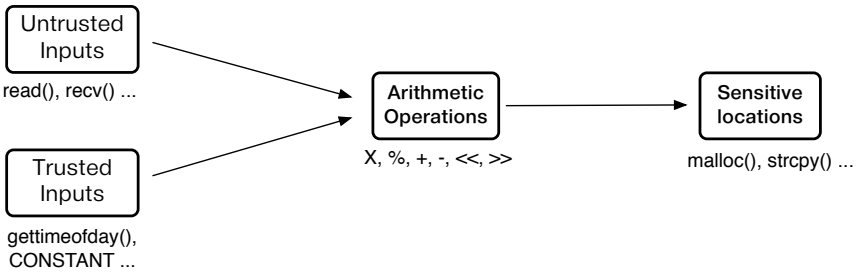

Figure 3.3: Information Flows to and from the Location of an Arithmetic Error.

when presenting results to the user. This partitioning can be achieved either statically or dynamically. In our prototype, we adopt static Information-Flow Tracking (IFT) [119, 85], to appropriately partition and report errors based on their criticality. An overall view of how IFT can by used towards this end is presented in Figure 3.3: arithmetic operations that result in integer errors are analyzed to determine if untrusted inputs affect or may affect the operation, or if the arithmetic operation's result propagates (or may propagate[3]) into for sensitive locations. With respect to criticality, highest priority is given to errors propagating into sensitive operations, followed by errors with variables originating from untrusted inputs. Trusted sources and well known coding constructs (such as constant assignments that may result in an overflow) are considered as low priority errors.

3.3.2.3 Implementation & Evaluation

We prototype the aforementioned design in IntFlow, a framework build on top of the LLVM [97] compiler. In particular, IntFlow extends the IOC [53] integer error detection toolchain to achieve context-aware reporting. The architecture of our prototype is depicted in Figure 3.4: IntFlow's two main components are an integer error detection engine module (IOC [53]) as well an information flow tracking module

[3]due to the nature of static IFT, it is often infeasible to determine at compile time if an error actually affects a subsequent operation. This use of static IFT is not without limitations: long call chains could impact the effectiveness of IntFlow at reducing false positives. However, alternate methods could be used to perform the desired partitioning of errors. Such methods could combine, for instance, runtime monitors with dynamic flow tracking.

(`llvm-deps` [119]), which can be fine-tuned through user-provided configuration files. During compilation, arithmetic error checks are inserted by IOC at all points involving integer operations. These checks are subsequently selectively filtered by IntFlow to determine any errors that may occur at runtime fall within known code constructs (such as constant assignments), whether they involve variables that originate from trusted or untrusted inputs, or whether the operation affects sensitive operations. Once this filtering is complete, IntFlow creates a report for the analyst, denoting the context in which the respective error occured (e.g., if it affects a sensitive operation).

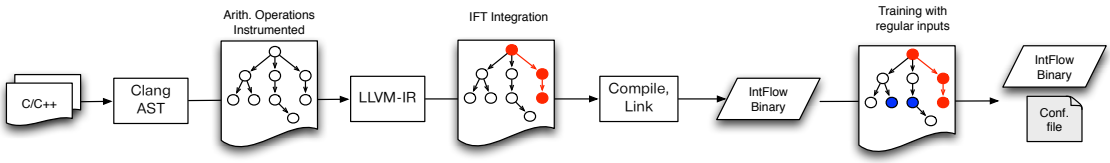

Figure 3.4: IntFlow's Architecture.

IntFlow is implemented as an LLVM [97] pass consisting of ∼3,000 lines of C++ code.[4] This pass is placed at the earliest stage of the LLVM pass dependency tree to prevent subsequent optimization passes from optimizing away any critical integer operations.

We evaluated IntFlow on both artificial and real-world vulnerabilities [140, 85] Artificial vulnerabilities corresponding to various types of the Common Weakness Enumeration (CWE) [49] system were inserted to a set of real-world applications. This set of applications was independently provided by the MITRE organization [14]. For evaluating IntFlow over real-world vulnerabilities, we used four widely-used applications and analyzed whether IntFlow detects known integer-related CVEs included in these programs. The programs under scrutiny where Dillo [54], GIMP [62], SWFTools [173] and Pidgin [139], and the respective CVEs corresponded to integer

[4] IntFlow can be invoked by simply passing the appropriate flags to the compiler, without any further action needed from the side of the developer. Although IOC has been integrated into the LLVM main branch since version 3.3, for the current prototype of IntFlow we used an older branch of IOC that supports a broader set of error classes than the latest one.

overflows and signedness errors [140]. To collect error reports, we ran each application with benign inputs as follows: for Gimp, we scaled a sample image and exported it as GIF [140]; for SWFTools, we used the `pdf2swf` utility with a popular e-book as input; for Dillo, we visited a webpage and downloaded content [140]; and for Pidgin, we performed various common tasks such as registering a new account and logging in and out of the service. Finally, we evaluated the effectiveness of IntFlow's IFT analysis in reducing false positives by running IntFlow on the SPEC CPU2000 benchmarking suite and comparing its reported errors with those of IOC. The above experiments demonstrated that IntFlow achieves up to 89% suppression of bening error reports over standalone static code instrumentation, without a negative impact on non-bening errors.

3.3.3 Discussion

In the previous Section we examined how compiler analyses can become more targeted providing contextual information when detecting given classes of errors. To demonstrate this point, we described the design of IntFlow, which reports integer errors with an accompanying criticality score and drastically reduces false positives so that developers are able to prioritize the respective fixes accordingly.

IntFlow employs static infromation flow tracking for to augment the compiler's passes and partitions the respective errors based on their semantic properties. However, similar analyses can be performed via different techniques, such as by deploying dynamic information flow tracking, hypervisors or compiler-inserted runtime monitors. Additionally, in the more general case of feedback-driven testing frameworks, context-aware partitioning of states can be used not only to achieve more selective targeting of errors of a given type but also as a means of re-using existing infrastructure to target broader classes of bugs. To make this point clearer, let us consider the design depicted in Figure 3.5 which is typical of common feedback-driven frameworks

like concolic testing toolchains [67, 168] and fuzzers [206, 105].

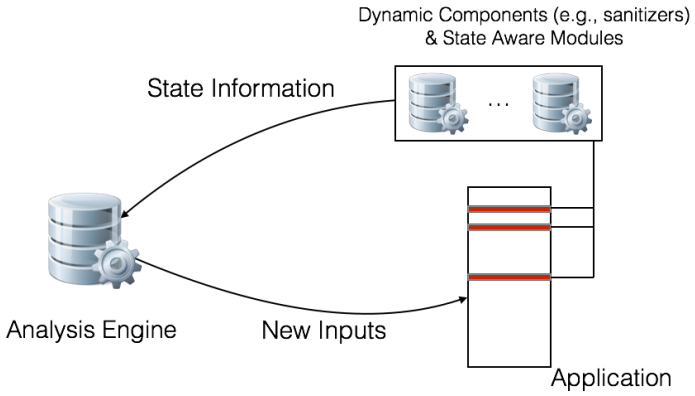

Figure 3.5: Typical Architecture for Feedback-driven Testing Frameworks.

In a typical compiler-assisted feedack-driven framework, the compiler inserts instrumentation into the compiled binary (highlighted in red in Figure 3.5), which is then used by dynamic components and state-aware modules (coverage buffers, sanitizers , etc.) to compute the state in which the tested application is in, when executing a particular input. This state information is passed into the analysis engine, which then computes subsequent inputs to be passed into the binary under scrutiny. In recent years, more and more toolchains deploy evolutionary or generational techniques for their input creation: if an input is deemed useful for testing, it is preserved as part of an active input corpus, together with other successful inputs. Inputs in the corpus are mutated and combined in order to produce future generations. As such, if the analysis engine prioritizes inputs that test particular portions or properties of the application, it is possible to steer the evolutionary input generation appropriately without implementing an input generation strategy from scratch. Instead, by combining a context-agnostic, generic mechanism (such as the evolutionary engine or the compiler passes) with context-aware modules (the fitness functions in use, taint-aware modules or context-aware compiler analyses), it is possible to re-use existing frameworks to target new types of bugs. In the next Chapters we will elaborate on how this can be achieve, presenting appropriate examples. Particularly, we will discuss

how context-aware guidance can aid in retrofitting fuzzers that traditionally target crash-inducing bugs to also address different classes of errors such as logic bugs or compilexity vulnerabilities.

Chapter 4

Adaptive Differential Testing

In the previous Chapter, we elaborated on how we can utilize context-aware analyses to target errors at a finer granularity. However, as discussed, the same technique can be used to retrofit feedback-driven frameworks so that the latter may target broader classes of errors. In this Chapter, as well as in Chapter 5, we will present two such prototype implementations built on top of state-of-the-art fuzzers.

4.1 Motivation

As we outlined in the Introduction, modern binary application testing frameworks predominantly focus on detection of memory safety violations. Binary application fuzzers, which are widely used by the application testing community, are perhaps the most typical example of this, since they almost solely focus on crash-inducing bugs. Other types of bugs, such as logic errors and semantic bugs, although equally important, are usually out-of-scope. However, semantic bugs are particularly dangerous for security-sensitive programs that are designed to classify inputs as either valid or invalid according to certain high-level specification. For instance, malware detectors are required to parse different file formats according to their respective specification, whereas libraries implementing particular RFCs (such as SSL/TLS libraries

handling X.509 certificates), need to conform to multiple, complex requirements. If semantic errors are present in security-critical software such as the above, attackers can exploit existing discrepancies between the implementation and its respective specification and cause the software to misbehave: in the case of malware detectors, attackers may mount evasion attacks, whereas in the case of SSL/TLS implementations, they may compromise the security guarantees of the respective connections by making the libraries accept invalid certificates.

In order to be able to detect deviations from a given specification or expected handling logic, however, it is necessary to have a point of reference, which will act as the ground-truth provider. Traditionally, a popular technique to use towards this end is differential testing [116, 202, 24]: differential testing uses similar programs as cross-referencing oracles to find semantic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Even if different applications are not available, it is possible to perform differential testing using different versions of the same application (e.g., before and after a given patch). Unfortunately, existing tools are domain-specific and inefficient, requiring large numbers of test inputs to find a single bug.

In this Chapter, we will examine how to utilize context-aware guidance to achieve differential fuzzing so as to discover program discrepancies in this setting, i.e., where at least one test program validates and accepts an input and another program with similar functionality rejects the same input as invalid. Particularly, we will demonstrate how, switching from a monolithic guidance engine based on code coverage, it will be possible to re-use a fuzzer's core components to target logic bugs additionally to crash-inducing vulnerabilities. A core observation motivating our work is that, in cases where execution is expected to conform to a particular logic, any deviations from the behavior dictated by that logic will somehow become observable throughout the program execution. Whether in the form error codes or messages, side effects or

via deviations in the expected control and data flow, *context-specific information is present in the execution*. Thus, if the analysis engine is able to partition the states in a way such as to steer the testing towards locating the source of the deviation, it will be able to detect semantic discrepancies such as the ones described above.

4.1.1 Example Use-case

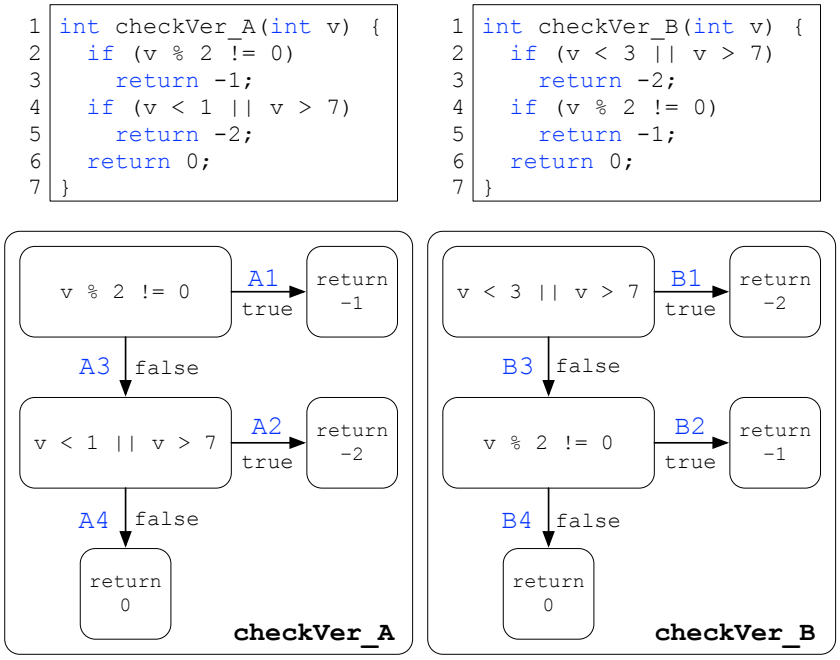

Figure 4.1: (*Top*) Simplified example of a semantic discrepancy and (*Bottom*) the corresponding simplified Control Flow Graphs.

To demonstrate the basic principles of our approach, let us consider the following example: suppose A and B are two different programs with similar functionality and that `checkVer_A` and `checkVer_B` are the functions validating the version number of the input files used by A and B respectively, as shown in Figure 4.1. Both of these functions return 0 to indicate a valid version number or a negative number (-1 or -2) to indicate an error. While almost identical, the two programs have a subtle discrepancy in their validation behavior. In particular, `checkVer_A` accepts an input

of `v == 2` as valid while `checkVer_B` rejects it with an error code of `-2`.

The above example, albeit simplified, is similar to semantic bugs found in deployed, real-world applications. However, this discrepancy *cannot* be found by individually testing each program without a formal specification, as it does not result in any explicitly erroneous behavior like a crash or assertion failure. Moreover, even if the above discrepancy constitutes a bug, it is hard for a state-of-the-art evolutionary coverage-based fuzzer to steer its input generation towards triggering this difference, since coverage is used in a context-agnostic manner. Thus, the fuzzer is agnostic to the semantic properties of the code being covered, and, similarly to how, in the example presented in Section 3.3, integer errors were reported by compilers without contextual information with respect to the context they appeared in, whether an input is more likely to trigger a logic error is not exposed to the analysis engine.

Similarly to our approach in Chapter 3, our key intuition is to augment existing fuzzer analyses with context-aware guidance so as to enable the fuzzer to perform differential, context-aware testing across several applications, steering its input generation towards the binaries' portions that are more likely to have a deviation in their expected behavior. This approach is expected to be successful in providing context-aware guidance since simultaneously testing multiple programs on the same input offers a plethora of information that can be used to compare the tested programs' behaviors relative to each other. Such examples include error messages, debug logs, rendered outputs, return values, observed execution paths of each program, etc. In the rest of this Chapter we will demonstrate that semantic discrepancies across programs of the same functionality,[1] are more likely to occur for the inputs that cause relative variations of features like the above.

[1]Similarly, across different versions of the same program.

4.1.2 Differential Diversity

Depending on the context-specific attributes of interest, we want to perform a *context-aware partitioning of states* during the testing session, and based on these states prioritize our input generation accordingly. Particularly, given n context-specific attributes $A = \{a_1, .., a_n\}$, we specify as a diversity tuple the tuple $< m_1, ..., m_n >$ where $m_i = f(a_i), i \in 1..n$ are context-specific metrics computed via some relation f over the attributes A. For instance, if we partition the edges of an application into three sets e_1, e_2, e_3 we can specify a coverage-diversity tuple as the portion of the edges of each group that have been accessed at one input execution. Thus, c_i denotes the sets of edges of e_i that were accessed throughout the execution of a given input, we can denote the edges covered across all groups for each input execution as $\{c_1, c_2, c_3\}$ and express the respective coverage diversity tuple as $< ||c_1||/||e_1||, ||c_2||/||e_2||, ||c_3||/||e_3|| >$. Using such diversity tuples, we can construct higher-level context-aware guidance engines. For instance, we can utilize the cardinality of the set of the different diversity tuples seen throughout the execution to determine if an input exhibits a behavior that we did not encounter in previous execution runs. In general, utilizing the above methodology, we may summarize behavioral assymetries of multiple tested programs, by creating the appropriate differential diversity (δ-diversity) metrics. In δ-diversity-based guidance, tracing is generalized across multiple programs, with individual program traces/behaviors being examined relative to each other, not in isolation, for guided input generation

Program behaviors can be summarized in different ways, e.g., in either a black-box (based on program log/warning/error messages, program outputs , etc.) or gray-box (e.g., program paths taken during execution) manner. In the following, we will examine examples of how to construct such context-aware δ-diversity engines in real-world scenarios. In particular, we will present NEZHA, the first, to the best of our knowledge, differential generic fuzzer, which utilizes δ-diversity-based context-aware guid-

ance to target both crash-inducing errors and logic bugs. Adopting an evolutionary algorithm approach, NEZHA begins with a corpus of seed inputs, applies mutations to each input in the corpus, and then selects the best-performing inputs for further mutations. The fitness of a given input for the testing process is determined based on the diversity it introduces in the observed context-aware attributes across the tested programs. Thus, if the chosen context-specific attribute to be used is the signals emitted by different applications or by different parts of the same application, inputs that generate the most diverse sets of signals (observed throughout the testing session across all applications), will receive higher priority than those that result in previously observed patterns.

4.1.3 Example: Gray-box Guidance

If program instrumentation is a feasible option, we can collect detailed runtime execution information from the test programs, for each input. For instance, knowledge of the portions of the Control Flow Graph (CFG) that are accessed during each program execution, can guide us into only mutating the inputs that are likely to visit new edges in the CFG. An edge in a CFG exists between two basic blocks if control may flow from one basic block to the other (e.g., A_1 is an edge in the simplified CFG for `checkVer_A` as shown in Figure 4.1). We illustrate how this information can be collectively tracked across multiple programs revisiting the example of Figure 4.1.

Table 4.1: A semantic bug that is missed by differential testing using code coverage but can be detected by NEZHA's path δ-diversity (gray-box) during testing of the examples shown in Figure 4.1. NEZHA's black-box δ-diversity input generation scheme (not shown in this example) would also have found the semantic bug.

			Execution Paths				**Add to Corpus**	
Gen.	**Mut.**	**Input**	`A`	`B`	Path Tuple	**δ-diversity**	Coverage	δ-diversity
seed	-	7	$\{A_1\}$	$\{B_3, B_2\}$	$P_1 = \langle\{A_1\}, \{B_3, B_2\}\rangle$	$\{P_1\}$	√	√
seed	-	0	$\{A_3, A_2\}$	$\{B_1\}$	$P_2 = \langle\{A_3, A_2\}, \{B_1\}\rangle$	$\{P_1, P_2\}$	√	√
seed	-	1	$\{A_1\}$	$\{B_1\}$	$P_3 = \langle\{A_1\}, \{B_1\}\rangle$	$\{P_1, P_2, P_3\}$	x	√
1	increment	2	$\{A_3, A_4\}$	$\{B_1\}$	$P_4 = \langle\{A_3, A_4\}, \{B_1\}\rangle$	$\{P_1, P_2, P_3, P_4\}$	-	√

Suppose that our initial corpus of test files (seed corpus) consists of three input files, with versions 7, 0, and 1 ($I_0 = \{7, 0, 1\}$). We randomly extract one input from I_0 to start our testing: suppose the input with `v=7` is selected and then passed to both `checkVer_A` and `checkVer_B`. As shown in Table 4.1, the execution paths for programs A and B (i.e., the sequence of unique edges accessed during the execution of each program) are $\{A_1\}$ and $\{B_3, B_2\}$ respectively. The number of edges covered in each program is thus 1 and 2 for A and B respectively, whereas the coverage achieved across both programs is 1+2 = 3. One may drive the input generation process favoring the mutation of inputs that increase coverage (i.e., exercise previously unexplored edges). Since `v=7` increased the code coverage, it is added to the corpus that will be used for the next generation: $I_1 = \{7\}$. In the following stage of the testing, we pick any remaining inputs from the current corpus and pass them to programs A and B. Selecting `v=0` as the next input will also increase coverage, since execution touches three previously-unseen edges (A_3, A_2 and B_1), and thus the file is picked for further mutations: $I_1 = \{7, 0\}$. At this stage, the only input of I_0 that has not been executed is `v=1`. This input's execution does not increase coverage, since both edges A_1 and B_1 have been visited again, and thus `v=1` is not added to I_1 and will not be considered for future mutations. However, we notice that `v=1`, with a single increment mutation, could be transformed to an input that would disclose the discrepancy between programs A and B, had it not been discarded. This example demonstrates that simply maximizing edge-coverage often *misses* interesting inputs that may trigger semantic bugs. By contrast, had we tracked the δ-diversity using path tuples across past iterations, input `v=1` would invoke the path tuple $\langle\{A_1\}, \{B_1\}\rangle$, which, as a *pair/combination*, would have not been seen before. Thus, using a *path δ-diversity* state, instead of code coverage, results in `v=1` been considered for further mutations. As seen in Table 4.1, the mutated input `v=2` uncovers the semantic bug.

4.1.4 Example: Black-box Guidance

If program instrumentation or binary rewriting are not feasible options, we may still adapt the notion of program diversity to a black-box setting. The key intuition is, again, to look for previously unseen *patterns* across the observed outputs of the tested programs. Depending on the context of the application being tested, available outputs may vary greatly. For instance, a malware detector may only provide one bit of information based on whether some input file contains a malware or not, whereas other applications may offer richer sets of outputs such as graphical content, error or debug messages, values returned to the executing shell, exceptions, etc. In the context of differential testing, the outputs of a single application A can be used as a reference against the outputs of all other applications being tested. For example, if browsers A, B, and C are differentially tested, one may use browser A as a reference and then examine the contents of different portions of the rendered Web pages with respect to A, using an arbitrary number of values for the encoding (different values may denote a mismatch in the CSS or HTML rendering, etc.).

Regardless of the output formulation, however, for each input used during testing, NEZHA may receive a corresponding set of *output values* and then only select the inputs that result in new output tuples for further mutations. In the context of the example of Figure 4.1, let us assume that the outputs passed to NEZHA are the values returned by routines `checkVer_A` and `checkVer_B`. If inputs $0, 7$, and 1 are passed to programs A and B, NEZHA will update its internal state with all unique output tuples seen so far: $\{\langle -1, -1\rangle, \langle -2, -2\rangle, \langle -1, -2\rangle\}$. Any new input which will result in a previously unseen tuple will be considered for future mutations, otherwise it will be discarded (e.g., with the aforementioned output tuple set, input 2 resulting in tuple $\langle 0, -2\rangle$ would be considered for future mutations, but input 9 resulting in $\langle -1, -2\rangle$ would be discarded).

4.2 NEZHA

4.2.1 Design

NEZHA is input-format-agnostic and can optionally use a set of initial seed inputs to bootstrap the input generation process. Note that the seed files themselves do not need to trigger any semantic bugs. We empirically demonstrate that NEZHA can efficiently detect subtle semantic differences in large, complex, real-world software. In particular, we use NEZHA for testing: (i) the ELF and XZ file parsing in two popular command-line applications and the ClamAV malware detector, (ii) X.509 certificate validation across six major SSL/TLS libraries and (iii) PDF parsing/rendering in three popular PDF viewers. NEZHA discovered 778 distinct discrepancies across all tested families of applications, many of which constitute previously unknown security vulnerabilities. For example, we found two evasion attacks against ClamAV, one for each of the ELF and XZ parsers. Moreover, NEZHA was able to pinpoint 14 unique differences even among forks of the same code base like the OpenSSL, LibreSSL, and BoringSSL SSL/TLS implementations.

In each testing session, NEZHA observes the relative behavioral differences across all tested programs to maximize the number of reported semantic bugs. To do so, NEZHA uses Evolutionary Testing (ET) [131], inferring correlations between the inputs passed to the tested applications and their observed behavioral asymmetries, and, subsequently, refines the input generation, favoring more promising inputs. Contrary to existing differential testing schemes which drive their input generation using *monolithic* metrics such as the code coverage that is achieved across the tested applications, NEZHA utilizes the novel concept of δ-diversity: metrics that preserve the *differential diversity* (δ-diversity) of the tested applications will perform better at finding semantic bugs than metrics that overlook relative asymmetries in the applications' execution. The motivation behind δ-diversity becomes clearer if we examine

Algorithm 1 DiffTest: Report all discrepancies across applications $\mathcal{A}$ after n generations, starting from a corpus $\mathcal{I}$

```
1:  procedure DiffTest(I, A, n, GlobalState)
2:      discrepancies = ∅ ;reported discrepancies
3:      while generation ≤ n do
4:          input = RandomChoice(I)
5:          mut_input = Mutate(input)
6:          generation_paths = ∅
7:          generation_outputs = ∅
8:          for app ∈ A do
9:              app_path, app_outputs = Run(app, mut_input)
10:             genereneration_paths ∪= {app_path}
11:             genereneration_outputs ∪= {app_outputs}
12:         end for
13:         if NewPattern(generation_paths,
                          generation_outputs,
                          GlobalState) then
14:             I ← I ∪ mut_input
15:         end if
16:         if IsDiscrepancy(generation_outputs) then
17:             discrepancies ∪= mut_input
18:         end if
19:         generation = generation + 1
20:     end while
21:     return discrepancies
22: end procedure
```

the following example. Suppose we are performing differential testing between applications A and B. Now, suppose an input I_1 results in a combined coverage C across A and B, exercising 30% of the CFG edges in A and 10% of the edges in B. A different input I_2, that results in the same overall coverage C, however exercising 10% of the edges in A and 28% of the edges of B, would not be explored further under monolithic schemes, despite the fact that it exhibits much different behavior in each individual application compared to input I_1.

We present Nezha's core engine in Algorithm 1. In each testing session, Nezha examines if different inputs result in previously unseen relative execution *patterns* across the tested programs. Nezha starts from a set of initial seed inputs $\mathcal{I}$, and

performs testing on a set of programs $\mathcal{A}$ for a fixed number of generations (n). In each generation, NEZHA randomly selects (line 4) and mutates (line 5), one input (individual) out of the population $\mathcal{I}$, and tests it against each of the programs in $\mathcal{A}$. The recorded execution paths and outputs for each application are added to the sets of total paths and outputs observed during the current generation (lines 8-12). Subsequently, if NEZHA determines that a *new execution pattern* was observed during this input execution, it adds the respective input to the input corpus, which will be used to produce the upcoming generation (lines 13-14). Finally, if there was a discrepancy in the outputs of the tested applications, NEZHA adds the respective input to the set of total discrepancies found (lines 16-18). Whether a discrepancy is observed in each generation depends on the outputs of the tested programs: if at least one application rejects an input and at least one other accepts it, a discrepancy is logged.

4.2.2 δ-diversity Guidance

In Algorithm 1, we demonstrated that NEZHA adds an input to the active corpus only if that input exhibits a newly seen pattern. In traditional evolutionary algorithms, the fitness of an individual for producing future generations is determined by its fitness score. In this section, we explain how δ-diversity can be used in NEZHA's guidance engines, both in a gray-box and a black-box setting.

4.2.2.1 Gray-box Guidance

The most prevalent guidance mechanism in gray-box testing frameworks is the code coverage achieved by individual inputs across the sets of tested applications. Code coverage can be measured using function coverage (i.e., the functions accessed in one execution run), basic block coverage or edge coverage. However, as discussed previously, this technique is not well suited for finding semantic bugs. By contrast,

NEZHA leverages relative asymmetries of the executed program paths to introduce two novel δ-diversity *path selection* guidance engines, suitable for efficient differential testing.

Suppose a program p is executing under an input i. We call the sequence of edges accessed during this execution the *execution path* of p under i, denoted by $path_{p,i}$. Tracking all executed paths (i.e., all the sequences of edges accessed in the CFG) is impractical for large-scale applications containing multiple loops and complex function invocations. In order to avoid this explosion in tracked states, NEZHA's gray-box guidance uses two different approximations of the execution paths, one of coarse granularity and the other offering finer tracking of the relative execution paths.

Path δ-diversity (coarse): Given a set of programs $\mathcal{P}$ that are executing under an input i, let $PC_{\mathcal{P},i}$ be the *Path Cardinality* tuple $\langle |path_{p_1,i}|, |path_{p_2,i}|, ..., |path_{p_{|\mathcal{P}|},i}| \rangle$. Each $PC_{\mathcal{P},i}$ entry represents the total number of edges accessed in each program $p_k \in \mathcal{P}$, for one *single* input i. Notice that $PC_{\mathcal{P},i}$ differs from the total coverage achieved in the execution of programs $\mathcal{P}$ under i, in the sense that $PC_{\mathcal{P},i}$ does not maintain a global, monolithic score, but a per-application count of the edges accessed, when each program is executing under input i. Throughout an entire testing session, starting from an initial input corpus $\mathcal{I}$, the overall (coarse) path δ-diversity achieved is the cardinality of the set containing all the above tuples: $PD_{Coarse} = |\bigcup_{i \in \mathcal{I}} \{PC_{\mathcal{P},i}\}|$.

This representation expresses the maximum number of unique path cardinality *tuples* for all programs in $\mathcal{P}$ that have been seen throughout the session. However, we notice that, although the above formulation offers a semantically richer representation of the execution, compared to total edge coverage, it constitutes a coarse approximation of the (real) execution paths. A finer-grained representation of the execution can be achieved if we take into account *which* edges, specifically, have been accessed.

Path δ-diversity (fine): Consider the path $path_{p,i}$, which holds all edges ac-

cessed during an execution of each program $p_k \in \mathcal{P}$ under input i. Let $path\_set_{p,i}$ be the set consisting of all unique edges of $path_{p,i}$. Thus $path\_set_{p,i}$ contains no duplicate edges, but instead holds only the CFG edges of p that have been accessed *at least once* during the execution. Given a set of programs $\mathcal{P}$, the (fine) path diversity of input i across $\mathcal{P}$ is the tuple $PD_{\mathcal{P},i} = \langle path\_set_{p_1,i}, path\_set_{p_2,i}, ..., path\_set_{p_{|\mathcal{P}|},i} \rangle$. Essentially, $PD_{\mathcal{P},i}$ acts as a "fingerprint" of the execution of input i across all tested programs and encapsulates relative differences in the execution paths *across* applications. For an entire testing session, starting from an initial input corpus $\mathcal{I}$, the (fine) path δ-diversity achieved is the cardinality of the set containing all the above tuples: $PD_{Fine} = |\bigcup_{i \in \mathcal{I}} \{PD_{\mathcal{P},i}\}|$.

To demonstrate how the above metrics can lead to different discrepancies, let us consider a differential testing session involving two programs A and B. Let A_n, B_n denote edges in the CFG of A and B, respectively, and let us assume that a given test input causes the paths $\langle A_1, A_2, A_1 \rangle$ and $\langle B_1 \rangle$ to be exercised in A and B respectively. At this point, $PD_{Coarse} = \{\langle 3, 1 \rangle\}$, and $PD_{Fine} = \{\langle \{A_1, A_2\}, \{B_1\} \rangle\}$. Suppose we mutate the current input, and the second (mutated) input now exercises paths $\langle A_1, A_2 \rangle$ and $\langle B_1 \rangle$ across the two applications. After the execution of this second input, PD_{Fine} remains unchanged, because the tuple $\langle \{A_1, A_2\}, \{B_1\} \rangle$ is already in the PD_{Fine} set. Conversely, PD_{Coarse} will be updated to $PD_{Coarse} = \{\langle 3, 1 \rangle, \langle 2, 1 \rangle\}$. Therefore, the new input will be considered for further mutation under a coarse path guidance, since it increased the cardinality of the PD_{Coarse} set, however it will be rejected under fine δ-diversity guidance. Finally, note that if we use total edge coverage as our metric for input selection, both the first and second inputs result in the same code coverage of 3 edges (two unique edges for A plus one edge for B). Thus, under a coverage-guided engine, the second input will be rejected as it does not increase code coverage, *despite* the fact that it executes in a manner that has not been previously observed across the two applications.

4.2.2.2 Black-box Guidance

As mentioned in Section 4.1.2, NEZHA's input generation can be driven in a black-box manner using any observable and countable program output, such as error/debug messages, rendered or parsed outputs, return values, etc. For many applications, especially those implementing particular protocols or RFCs, such outputs often uniquely identify deterministic execution patterns. For example, when a family of similar programs returns different error codes/messages, any change in one test program's returned error relative to the error codes returned by the other programs is highly indicative of the relative behavioral differences between them. Such output asymmetries can be used to guide NEZHA's path selection.

Output δ-diversity: Let p be a program which, given an input i, produces an output $o_{p,i}$. We define the output diversity of a family of programs $\mathcal{P}$, executing with a single input i, as the tuple $OD_{\mathcal{P},i} = \langle o_{p_1,i}, o_{p_2,i}, ..., o_{p_{|\mathcal{P}|},i} \rangle$. Across a testing session that starts from an input corpus $\mathcal{I}$, output δ-diversity tracks the number of unique output tuples that are observed throughout the execution of inputs $i \in \mathcal{I}$ across all programs in $\mathcal{P}$: $|\bigcup_{i \in \mathcal{I}} \{OD_{\mathcal{P},i}\}|$. Input generation based on output δ-diversity aims to drive the tested applications to result in as many different output combinations across the overall pool of programs, as possible. This metric requires no knowledge about the internals of each application and is completely black-box. As a result, it can even be applied on applications running on a remote server or in cases were binary rewriting or instrumentation is infeasible.

Output δ-diversity, as defined above, is not constrained solely to return values, error codes or other forms of inter-process communication messages. Instead, the same metric (or other metrics with similar characteristics), can be applied to *graphical* outputs as well. For instance, if to compare how different browsers render an HTML page, or how different image viewers render a particular image, we can define output diversity tuples by capturing the graphical output of each program and utilizing a

unique checksum - representative of the respective image that is displayed. Additionally, one may use finer-grained metrics for different outputs $o_{p_{|\mathcal{P}|},i}$, focusing on particular sub-components of the rendered graphical output, or on given attributes of interest (color accuracy, font/graphics rendering , etc.). What is formulated as "output" in each case, will also affect the quality of the respective guidance. We demonstrate in Section 4.2.4 that this black-box performs equally well as NEZHA's gray-box engines for programs that support fine-grained output values.

Algorithm 2 Determine if a new pattern has been observed

```
1:  procedure NewPattern(gen_paths, gen_outputs, GlobalState)
2:      IsNew =false
3:      if GlobalState.UsePDCoarse then
4:          IsNew |= PDCoarse(gen_paths, GlobalState)
5:      end if
6:      if GlobalState.UsePDFine then
7:          IsNew |= PDFine(gen_paths, GlobalState)
8:      end if
9:      if GlobalState.UseOD then
10:         IsNew |= OD(gen_outputs, GlobalState)
11:     end if
12:     return IsNew
13: end procedure
```

As described in Algorithm 1, whenever a set of applications is tested under NEZHA, a mutated input that results in a previously unseen pattern (Algorithm 1 - lines 13-15) is added to the active input corpus to be used in future mutations. Procedure `NewPattern` is called for each input (at every generation), after all tested applications have executed, to determine if the input exhibits a newly observed behavior and should be added in the current corpus. The pseudocode for the routine is described in Algorithm 2: for each of the active guidance engines in use, NEZHA calls the respective routine listed in Algorithm 3 and, if the path δ-diversity and output δ-diversity is increased for each of the modes respectively (i.e., the input results in a discovery of a previously unseen tuple), the mutated input is added to the current corpus.

Algorithm 3 NEZHA path selection routines

```
; Path δ-diversity (coarse)
; @generation_paths: paths for each tested app for current input
; @GS: GlobalState (bookkeeping of paths, scores, etc.)
procedure PDCoarse(generation_paths, GS)
    path_card = ∅
    for path in generation_paths do
        path_card ∪= {|path|}
    end for
    ; See if path_card tuple has been seen in the stored tuples of GlobalState
    new_card_tuple = {⟨path_card⟩} \ GS.PDC_tuples
    if new_card_tuple ≠ ∅ then
        ; If new, add to GlobalState and update score
        GS.PDC_tuples ∪= new_card_tuple
        GlobalState.PDC_Score = |GS.PDC_tuples|
        return true
    end if
    return false
end procedure

; Path δ-diversity (fine)
procedure PDFine(generation_paths, GS)
    path_set = ∅
    for path in generation_paths do
        path_set ∪= {path}
    end for
    new_paths = {⟨path_set⟩} \ GS.PDF_tuples
    if new_path_tuple ≠ ∅ then
        GS.PDF_tuples ∪= new_path_tuple
        GlobalState.PDF_Score = |GS.PDF_tuples|
        return true
    end if
    return false
end procedure

; Output δ-diversity
procedure OD(generation_outputs, GS)
    new_output_tuple = {⟨output_tuple⟩} \ GS.OD_tuples
    if new_output_tuple ≠ ∅ then
        GS.OD_tuples ∪= new_output_tuple
        GlobalState.OD_Score = |GS.OD_tuples|
        return true
    end if
    return false
end procedure
```

4.2.2.3 Automated Debugging

NEZHA is designed to efficiently detect discrepancies across similar programs. However, the larger the number of reported discrepancies and the larger the number of tested applications, the harder it is to identify unique discrepancies and to localize the root cause of each report. To aid bug localization, NEZHA stores each mutated input in its original form throughout the execution of each generation. NEZHA compares any input that caused a discrepancy with its corresponding stored copy (before the mutation occurred), and logs the difference between the two. As this input pair differs only on the part that introduced the discrepancy, the two inputs can subsequently be used for delta-debugging [208] to pinpoint the root cause of the difference. Finally, to aid *manual analysis* of reported discrepancies, NEZHA performs a bucketing of reported differences using the return values of the tested programs. Moreover, it reports the file similarity of reported discrepancies using context-triggered piecewise fuzzy hashing [94]. Automated debugging and bug localization in the context of differential testing is not trivial. Future additions in the current NEZHA design, as well as limitations of existing techniques are discussed further in Section 4.3.

4.2.3 Implementation

We present NEZHA's architecture in Figure 4.2. NEZHA consists of two main components: its core engine and its runtime library. The runtime library collects all information necessary for NEZHA's δ-diversity guidance and subsequently passes it to the core engine. The core engine then generates new inputs through mutations, and updates the input corpus based on its δ-diversity guidance. We implemented NEZHA using Clang v3.8. Our implementation consists of a total of 1545 lines of C++ code, of which 1145 correspond to the NEZHA core engine and 400 to NEZHA's runtime library.

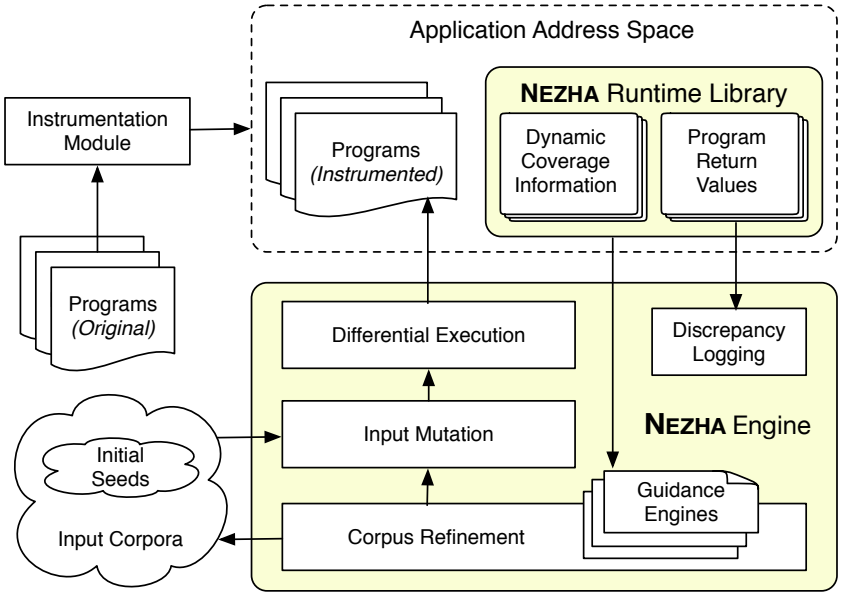

Figure 4.2: NEZHA Architecture.

4.2.4 Experimental Evaluation

In this section, we assess the effectiveness of NEZHA both in terms of finding discrepancies in security-critical, real-world software, as well as in terms of its core engine's efficiency compared to other differential testing tools. In particular, we evaluate NEZHA by differentially testing six major SSL libraries, file format parsers, and PDF viewers. We also compare NEZHA against two domain-specific differential testing engines, namely Frankencerts [22] and Mucerts [37], and two state-of-the-art domain-agnostic guided mutational fuzzers: American Fuzzy Lop (AFL) [206], and libFuzzer [105]. Our evaluation aims at answering the following research questions: 1) is NEZHA effective at finding semantic bugs? 2) does it perform better than domain-specific testing engines? 3) does it perform better than domain-agnostic coverage-guided fuzzers? 4) what are the benefits and limitations of each of NEZHA's δ-diversity engines?

4.2.4.1 Experimental Setup

X.509 certificate validation: We examine six major SSL libraries, namely OpenSSL (v1.0.2h), LibreSSL (v2.4.0), BoringSSL (f0451ca[2]), wolfSSL (v3.9.6), mbedTLS (v2.2.1) and GnuTLS (v3.5.0). Each of the SSL/TLS libraries is instrumented with SanitizerCoverage and AdressSanitizer so that NEZHA has access to the programs' path and output information. For each library, NEZHA invokes its built-in certificate validation routines and compares the respective error codes: if at least one library returns an error code on a given certificate whereas another library accepts the same certificate, this is counted as a discrepancy.

For our experiments, our pool of seed inputs consists of 205,853 DER certificate chains scraped from the Web. Out of these, we sampled certificates to construct 100 distinct groups of 1000 certificates each. Initially, *no certificate in any of the initial 100 groups introduced a discrepancy* between the tested applications thus all reported discrepancies in our results are introduced solely due to the differential testing of the examined frameworks.

ELF and XZ parsing: We evaluate NEZHA on parsers of two popular file formats, namely the ELF and the XZ formats. For parsing of ELF files, we compare the parsing implementations in the ClamAV malware detector with that of the `binutils` package, which is ubiquitous across Unix/Linux systems. In each testing session, NEZHA loads a file and validates it using ClamAV and `binutils` (the respective validation libraries are `libclamav` and `libbfd`), and either reports it as a valid ELF binary or returns an appropriate error code. Both programs, including all their exported libraries, are instrumented to work with NEZHA and are differentially tested for a total of 10 million generations. In our experiments, we use ClamAV 0.99.2 and `binutils` v.2.26-1-1_all. Our seed corpus consists of 1000 Unix malware files sampled from VirusShare [184] and a plain '*hello world*' program.

[2]This refers to a git commit hash from BoringSSL's master branch.

Similar to the setup for ELF parsing, we compare the XZ parsing logic of ClamAV and XZ Utils [201], the default Linux/Unix command-line decompression tool for XZ archive files. The respective versions of the tested programs are ClamAV 0.99.2 and `xzutils` v5.2.2. Our XZ seed corpus uses the XZ files from the XZ Utils test suite (a total of 74 archives) and both applications are differentially tested for a total of 10 million generations.

PDF viewers: We evaluate NEZHA on three popular PDF viewers, namely the Evince (v3.22.1), MuPDF (v1.9a) and Xpdf (v3.04) viewers. Our pool of tested inputs consists of the PDFs included in the Isartor [81] testsuite. All applications are differentially tested for a total of 10 million generations. During testing, NEZHA forks a new process for each tested program, invokes the respective binary through `execlp`, and uses the return values returned by the execution to the parent process to guide the input generation using its output δ-diversity. Determined based on the return values of the tested programs, the discrepancies constitute a conservative estimate of the total discrepancies, because while the return values of the respective programs may match, the rendered PDFs may differ.

All our measurements were performed on a system running Debian GNU/Linux 4.5.5-1 while our implementation of NEZHA was tested using Clang version 3.8.

4.2.4.2 Effectiveness in Discovering Discrepancies

The results of our analysis with respect to the discrepancies and memory errors found are summarized in Table 4.2. NEZHA found 778 validation discrepancies and 8 memory errors in total. Each of the reported discrepancies corresponds to a unique tuple of error codes, where at least one application accepts an input and at least another application rejects it. Examples of semantic bugs found are presented in Section 4.2.4.6.

We observe that, out of the total 778 discrepancies, 764 were reported during

Table 4.2: Result Summary for our Analysis of NEZHA.

Type	SSL Certificate	XZ Archive	ELF Binary	PDF File
Discrepancies	764	5	2	7
Errors & Crashes	6	2	0	0

our evaluation of the tested SSL/TLS libraries. The disproportionately large number of discrepancies found for SSL/TLS is attributed to the fine granularity of the error codes returned by these libraries, as well as to the larger number of applications being tested (six applications for SSL/TLS versus three for PDF and two for ELF/XZ).

To provide an insight into the impact that the number of tested programs has over the total reported discrepancies, we measure the total discrepancies observed between every *pair* of the six SSL/TLS libraries. In the pair-wise comparison of Table 4.3, two different return-value tuples that have the same error codes for libraries A and B are not counted twice for the (A, B) pair (i.e., we regard the output tuples $\langle 0, 1, 2, 2, 2, 2 \rangle$ and $\langle 0, 1, 3, 3, 3, 3 \rangle$ as one pairwise discrepancy with respect to the first two libraries). We observe that *even in cases of very similar code bases* (e.g., OpenSSL and LibreSSL which are forks of the same code base), NEZHA *successfully reports multiple unique discrepancies.*

Table 4.3: Number of *unique pairwise* discrepancies between different SSL libraries. Note that the input generation is still guided using *all* of the tested SSL/TLS libraries.

	LibreSSL	BoringSSL	wolfSSL	mbedTLS	GnuTLS
OpenSSL	10	1	8	33	25
LibreSSL	-	11	8	19	19
BoringSSL	-	-	8	33	25
wolfSSL	-	-	-	6	8
mbedTLS	-	-	-	-	31

The results presented in Table 4.2 are new reports and not reproductions of existing ones. They include multiple confirmed, previously unknown semantic error. Moreover, NEZHA was more efficient at reporting discrepancies than all guided or

unguided frameworks we compared it against (see Sections 4.2.4.3 & 4.2.4.4 for further details on this analysis). We present some examples of semantic bugs that have already been identified and patched by the respective software development teams in Section 4.2.4.6.

In addition to finding semantic bugs, NEZHA was equally successful in uncovering previously unknown memory corruption vulnerabilities and crashes in the tested applications. In particular, five of them were crashes due to invalid memory accesses (four cases in wolfSSL and one in GnuTLS), one was a memory leak in GnuTLS and two were use-after-free bugs in ClamAV.

4.2.4.3 Comparison with State-of-the-art Domain-specific Frameworks

One may argue that being domain-independent, NEZHA may not be as efficient as successful domain-specific frameworks. To address this concern, we compared NEZHA against Frankencerts [22], a popular black-box unguided differential testing framework for SSL/TLS certificate validation, as well as Mucerts [37], which builds on top of Frankencerts performing Markov Chain Monte Carlo (MCMC) sampling to diversify certificates using coverage information. Frankencerts generates mutated certificates by randomly combining X.509 certificate fields that are decomposed from a corpus of seed certificates. Despite its unguided nature, Frankencerts successfully uncovered a multitude of bugs in various SSL/TLS libraries. Mucerts adapt many of Frankencerts core components but also stochastically optimize the certificate generation process based on the coverage each input achieves in a single application (OpenSSL). Once the certificates have been generated from this single program, they are used as inputs to differentially test all SSL/TLS libraries.

To make a fair comparison between NEZHA, Frankencerts, and Mucerts, we ensure that all tools are given the same sets of input seeds. Furthermore, since Frankencerts is a black-box tool, we restrict NEZHA to only use its black-box output δ-diversity

guidance, across all experiments.

Since the input generation is stochastic in nature due to the random mutations, we perform our experiments with multiple runs to obtain statistically sound results. In particular, for each of the input groups of certificates we created (100 groups of 1000 certificates each), we generate 100,000 certificate chains using Frankencerts, resulting in a total of 10 million Frankencerts-generated chains. Likewise, passing as input each of the above 100 corpuses, we run NEZHA for 100,000 generations (resulting in 10 million NEZHA-executed inputs). Mucerts also start from the same sets of inputs and execute in mode 2, which according to [37] yields the most discrepancies with highest precision. We use the return value tuples of the respective programs to identify unique discrepancies (i.e., unique tuples of return values seen during testing).

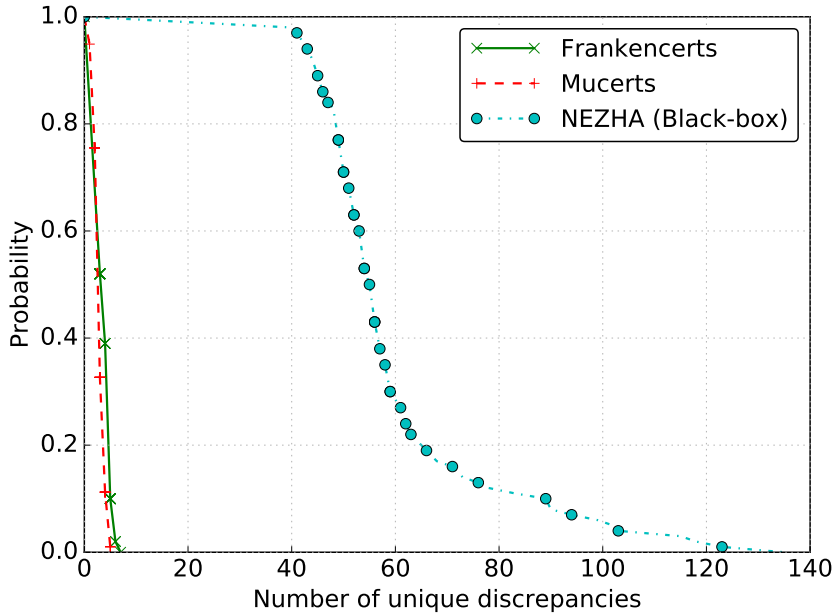

Figure 4.3: Probability of finding at least n unique discrepancies starting from the same seed corpus of 1000 certificates and running 100,000 iterations. The results are averages of 100 runs each starting with a different seed corpus.

We present the relative number and distribution of discrepancies found across Frankencerts, Mucerts and NEZHA in Figures 4.3 and 4.4. Overall, NEZHA reported 521 unique discrepancies, compared to 10 and 19 distinct discrepancies for

Frankencerts and Mucerts respectively. NEZHA reports 52 times and 27 times more discrepancies than Frankencerts and Mucerts respectively, *starting from the same sets of initial seeds* and running for the same number of iterations, achieving a respective coverage increase of 15.22% and 33.48%.

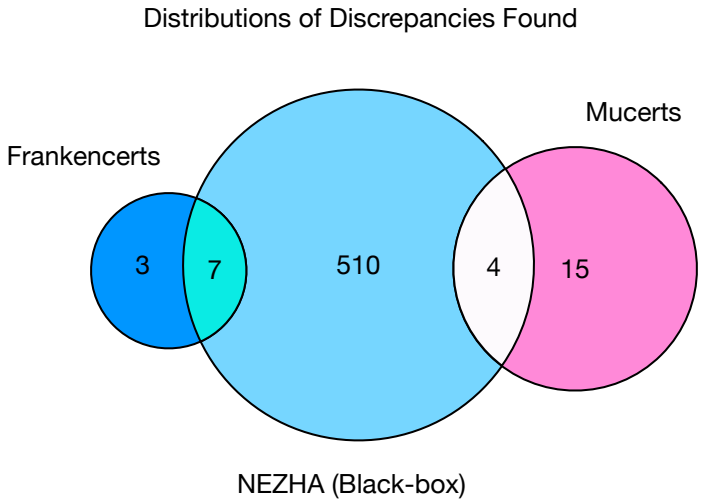

Figure 4.4: Unique discrepancies observed by Frankencerts, Mucerts and NEZHA (black-box). The results are averages of 100 runs each starting with a different seed corpus of 1000 certificates.

We observe that, while both Frankencerts and Mucerts reported a much smaller number of discrepancies than NEZHA, they found 3 and 15 discrepancies respectively that were missed by NEZHA. We posit that this is due to the differences in their respective mutation engines. Frankencerts and Mucerts start from a corpus of certificates, break all the certificates in the corpus into the appropriate fields (extensions, dates, issuer , etc.), then randomly sample and mutate those fields to merge them back together in *new* chains, however respecting the semantics of each field (for instance, Frankencerts might mutate and merge the extensions of two or three certificates to form the extensions field of a new chain but will not substitute a date field with an extension field). On the contrary, NEZHA performs its mutations sequentially, without mixing together different components of the certificates in the seed corpus, as it

does not have any knowledge of the input format.

It is noteworthy that, despite the fact that NEZHA's mutation operators are domain-independent, NEZHA's guidance mechanism allows it to favor inputs that are *mostly syntactically correct*. Compared to Frankencerts or Mucerts that mutate certificates at the granularity of X.509 certificate fields, without violating the core structure of a certificate, NEZHA still yields more bugs. Finally, when running NEZHA's mutation engine without any guidance, on the same inputs, we observe that no discrepancies were found. Therefore, NEZHA's efficacy in finding discrepancies can only be attributed to its black-box δ-diversity-based guidance.

4.2.4.4 Comparison with State-of-the art Coverage-guided Domain-independent Fuzzers

None of the state-of-the-art domain-agnostic fuzzers like AFL natively support differential testing. However, they can be adapted for differential testing by using them to generate inputs with a single test application and then invoking the full set of tested applications with the generated inputs. To differentially test our suite of six SSL/TLS libraries, we first generate certificates using a coverage-guided fuzzer on OpenSSL, and then pass these certificates to the rest of the SSL libraries, similar to how differential testing is performed by Mucerts. The discrepancies reported across all tested SSL libraries, if we run AFL (v. 2.35b)[3] and libFuzzer on a standalone program (OpenSSL) are reported in Figure 4.5. We notice that NEZHA yields 6 times and 3.5 times more differences per tested input, on average, than AFL and libFuzzer respectively.

This demonstrates that driving input generation with a single application is ill-suited for differential testing. In the absence of a widely-adopted domain-agnostic differential testing framework, we modified libFuzzer's guidance engine to support

[3]Since version 2.33b, AFL implements the `explore` schedule as presented in AFLFast [20], thus we omit comparison with the latter.

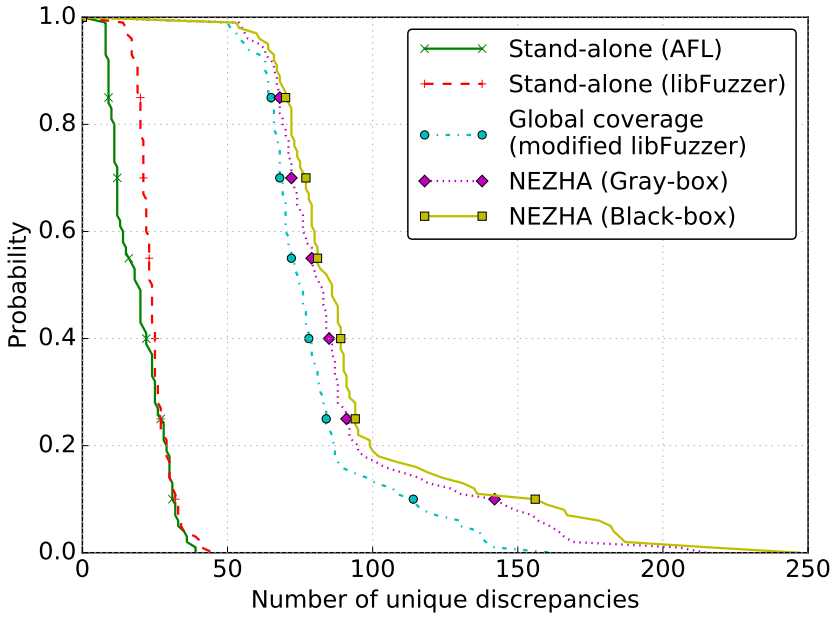

Figure 4.5: Probability of finding at least n unique discrepancies after $100,000$ executions, starting from a corpus of 1000 certificates. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

differential testing using *global* code coverage. Apart from its guidance mechanisms, this modified libFuzzer[4] is identical to NEZHA in terms of all other aspects of the engine (mutations, corpus minimization , etc.). Even so, as shown in Figure 4.5, NEZHA still yields 30% more discrepancies per tested input. Furthermore, NEZHA also achieves 1.3% more code coverage.

4.2.4.5 Engine Evaluation

To compare the performance of NEZHA's δ-diversity engines, we run NEZHA on the six SSL/TLS libraries used in our previous experiments, enabling a single guidance engine at a time. Before evaluating NEZHA's δ-diversity guidance, we ensured that the discrepancies reported are a result of NEZHA's guidance and not attributed to NEZHA's mutations. Indeed, when we use NEZHA without any δ-diversity guidance, *no* discrepancies were found across the SSL/TLS libraries.

[4]Corresponding git commit is 1f0a7ed0f324a2fb43f5ad2250fba68377076622

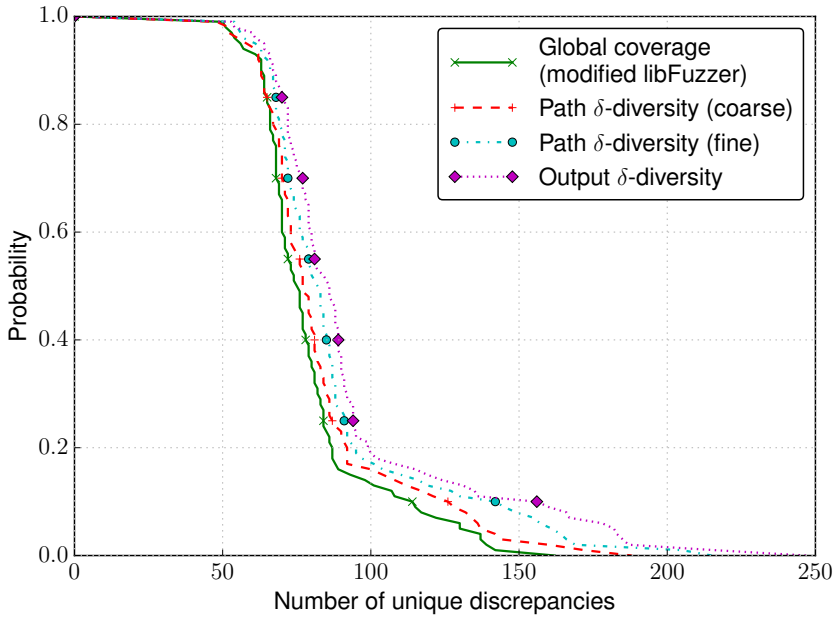

Figure 4.6: Probability of finding at least n unique discrepancies for each of NEZHA's δ-diversity engines after $100,000$ executions. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

Figures 4.6 and 4.7 show the relative performances of different δ-diversity engines in terms of the number of unique discrepancies they discovered. Figure 4.6 shows the probability of finding at least n unique discrepancies across the six tested SSL/TLS libraries, starting from a corpus of 1000 certificates and performing $100,000$ generations. For this experimental setting, we notice that NEZHA reports at least 57 discrepancies with more than 90% probability regardless of the engine used. Furthermore, all δ-diversity engines report more discrepancies than global coverage. Figure 4.7 shows the rate at which each engine finds discrepancies during execution. We observe that both δ-diversity guidance engines report differences at higher rates than global coverage using the same initial set of inputs.

Overall throughout this experiment, NEZHA's output δ-diversity yielded 521 discrepancies, while path δ-diversity yielded 491 discrepancies, resulting in 30% and 22.75% more discrepancies than using global code coverage to drive the input generation (global coverage resulted in 400 unique discrepancies). With respect to the

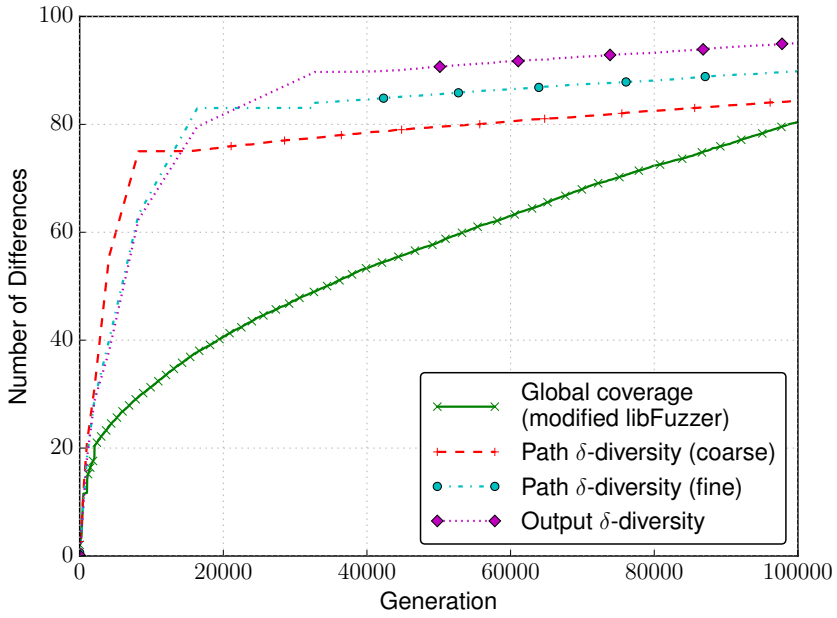

Figure 4.7: Unique discrepancies observed for each of NEZHA's δ-diversity engines per generation. The results are averages of 100 runs each starting from a different seed corpus of 1000 certificates.

coverage of the CFG that is achieved, output δ-diversity and path δ-diversity guidance achieves 1.38% and 1.21% higher coverage then global coverage guidance (graphs representing the coverage and population increase at each generation are presented in Section 4.2.4.5).

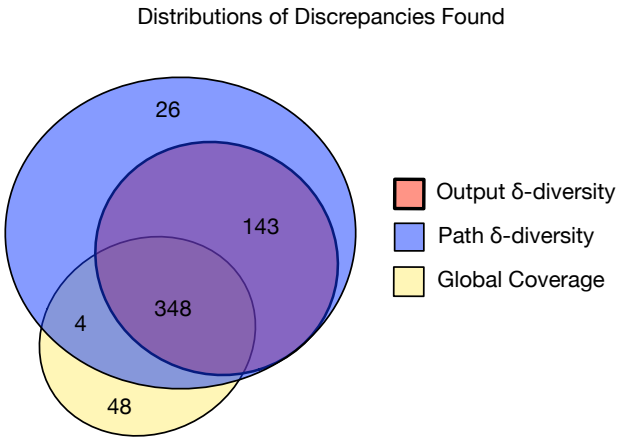

Figure 4.8: Distribution of bugs found by NEZHA's δ-diversity engines versus NEZHA using global-coverage-based guidance.

The distribution of the discrepancies reported by the different engines is presented in Figure 4.8. We notice that 348 discrepancies have been found by all three guidance engines, 121 discrepancies are reported using δ-diversity and 48 discrepancies are reported by our custom libFuzzer global code coverage engine. This result is a clear indication that δ-diversity performs differently than global code coverage with respect to input generation, generating a *broader* set of discrepancies for a *given time budget*, while exploring similar portions of the application CFG (1.21% difference in coverage for the same setup).

One notable result from this experiment is that output δ-diversity, despite being black-box, achieves equally good coverage with NEZHA's gray-box engines and even reports more unique discrepancies. This is a very promising result as it denotes that the internal state of an application can, in some cases, be adequately approximated based on its outputs alone assuming that there is enough diversity in the return values.

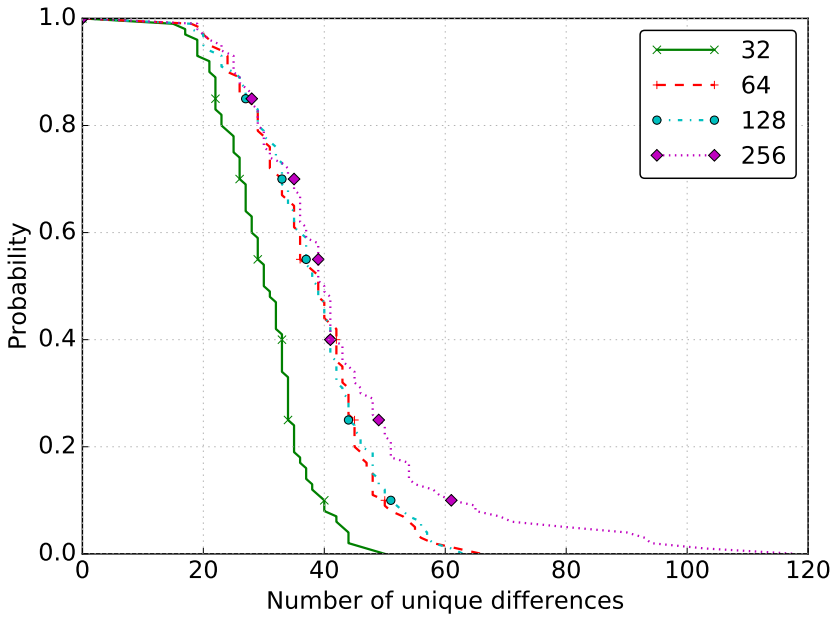

Figure 4.9: Probability of finding at least n unique discrepancies across OpenSSL, LibreSSL, and BoringSSL with NEZHA running under output δ-diversity, for varying numbers of error codes, after 100,000 executions (average of 100 runs, starting from a different seed corpus of 1000 certificates in each run).

However, we expect that output δ-diversity will perform worse for applications for which the granularity of the outputs is very coarse. For instance, the discrepancies that will be found in an application that provides debug messages or fine-grained error codes are expected to be more than those found in applications with less expressive outputs, (e.g., a web application firewall that only returns ACCEPT or REJECT based on its input). To verify this assumption, we perform an experiment with only three SSL libraries, i.e., OpenSSL, LibreSSL and BoringSSL, in which all libraries are only returning a *subset* of their supported error codes, namely at most $32, 64, 128$ and 256 error codes. Our results are presented in Figure 4.9. We notice that a limit of 32 error codes results in significantly fewer discrepancies than a more expressive set of error values. Finally, we should note that when we decreased this limit further, to only allow 16 possible error codes across all three libraries, Nezha did not find *any* discrepancies.

Coverage and population size for Nezha's different guidance engines: In Figures 4.10 and 4.11, we present the coverage and population increases for the different engines of Nezha for the experimental setup of Section 4.2.4.5.

We notice that δ-diversity engines converge as the generation numbers increase, however, in early stages of the testing process, the Path Cardinality and Path Diversity metrics show the greatest increase. We notice that out baseline metric, global edge coverage, shows marginally worse performance to this respect. A similar behavior can be seen in Figure 4.11, which shows the population size increase per generation, averaged across our 100 input sets, each consisting of 1000 certificates. As described in previous Sections, an individual is added to the population only if its respective fitness function sees an increase in diversity with respect to previous runs. This result, combined with the results of Figure 4.6 demonstrates that code coverage is not as good metric for differential testing as δ-diversity, since, not only does it not yield better results in terms of exploring new regions of the application, but it also

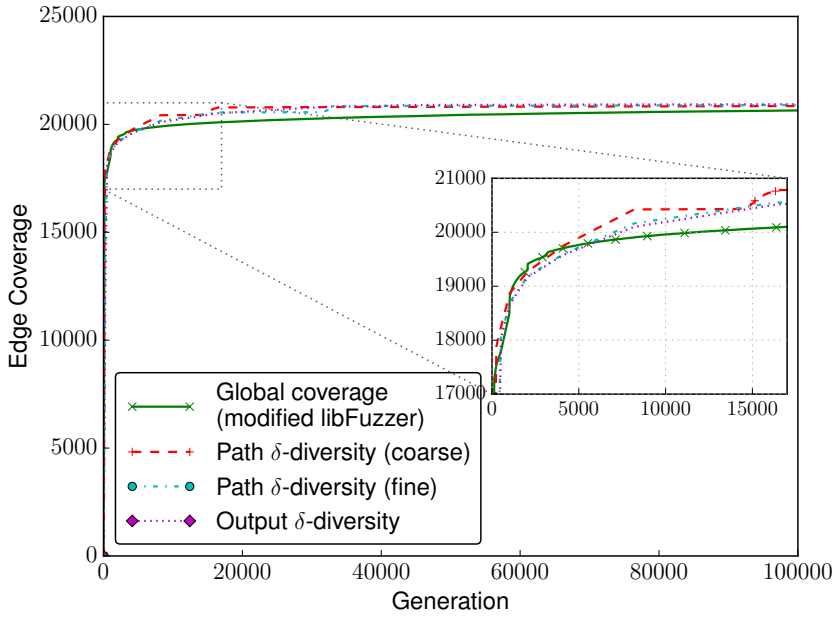

Figure 4.10: Coverage increase for each of NEZHA's engines per generation (average of 100 runs with a seed corpus of 1000 certificates).

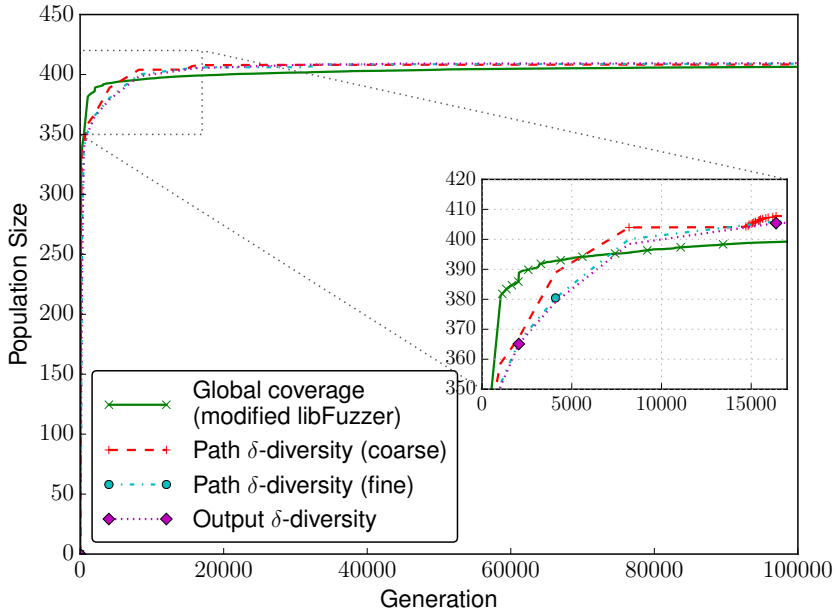

Figure 4.11: Population size increase for each of NEZHA's engines per generation (average of 100 runs, each starting from a seed corpus of 1000 certificates).

shows the worst rate in discovering new discrepancies, and the lower performance in expected number of discrepancies found.

4.2.4.6 Case Studies of Logic Errors

ClamAV errors: Discrepancies in the file format validation logic across programs can have dire security implications. Here we highlight two critical bugs, where ClamAV fails to parse specially crafted ELF and XZ files and thus does not scan them, despite the fact that the programs that commonly execute/extract these types of files process them correctly. These bugs allow an attacker to launch evasion attacks against ClamAV by injecting malware into specially crafted files.

ClamAV Mishandling of Malformed ELF Header: NEZHA uncovered a discrepancy in the way ClamAV and `binutils` handle the `EI_CLASS` field in the ELF header of Unix executables. This opens up the possibility of a critical evasion attack whereby a Unix malware with a corrupted ELF header can evade the detection of ClamAV, while retaining its capability to execute in the host OS. According to the ELF specifications, the `e_ident` member of the ELF `ElfXX_Ehdr` struct provides a means of encoding data for different processors and classes of machines and is the member of the ELF header that also contains the ELF magic bytes. `EI_CLASS` denotes whether a particular object file is 32-bit (0x1), 64-bit (0x2), or invalid (0x0). Values greater than 0x2 are illegal and, if checked properly, should result in an abortion of execution. When an ELF binary has the `EI_CLASS` field of the ELF identification indexes (`e_ident`) configured with an illegal value, ClamAV returns a `CL_EFORMAT` error when parsing the file and regards the file as an invalid ELF file. If ClamAV cannot infer the file format of the file in such situations, it skips the scanning of the file and flags the file as clean.

Despite ClamAV not properly parsing files with such malformed ELF headers, there is no corresponding check for this field when the executable file is invoked directly on the system. In particular, when the binary is executed with the `execve` invocation, the binary to be executed is passed to `load_elf_binary`,[5] in which only

[5]located in `binffmt_elf.c`

the first 4 magic bytes of the `e_ident` struct are checked. For this and subsequent checks to be performed, however, the portion of the binary that is expected to contain the ELF `ehdr` pointer, is first typecasted and copied to a struct inside the routine. Thus, if the segment and section headers, as well as other critical portions are not corrupted, they continue to hold the proper entries for symbol resolving to succeed. Thus, if the file is an executable (has an `e_type` of `ET_EXEC` or `ET_DYN`), all the segments of the file will be properly mapped to memory and execution will proceed.

```
1  static int cli_elf_fileheader(...) {
2    ...
3    switch(file_hdr->hdr64.e_ident[4]) {
4      case 1:
5        ...
6      case 2:
7        ...
8      default:
9        ...
10       return CL_EFORMAT;
```

Listing 4.1: ClamAV code that parses the `e_ident` field.

```
1  static int load_elf_binary(struct linux_binprm *bprm) {
2    ...
3    retval = -ENOEXEC;
4    if (memcmp(loc->elf_ex.e_ident, ELFMAG, SELFMAG) != 0)
5      goto out;
6    if (loc->elf_ex.e_type != ET_EXEC && loc->elf_ex.e_type != ET_DYN)
7      goto out;
8    if (!elf_check_arch(&loc->elf_ex))
9      goto out;
10   ...
```

Listing 4.2: Error checks for ELF loading in the Linux kernel (the `e_ident` field is not checked).

The mutational module of Nezha produces binaries that result in a parsing error in ClamAV, and yet can be successfully executed when invoked directly or

parsed through our `binutils` driver. To verify the exploitability of this bug, we also download a public Unix malware sample, corrupt the ELF header with a malformed `EI_CLASS`, and demonstrate that the malware can still be executed by the system, and yet does not get flagged by ClamAV as malicious.

ClamAV Mishandling of XZ Dictionary Size Field: According to the XZ specifications [179], the LZMA2 decompression algorithm in an archive can use a dictionary size ranging from 4kB to 4GB. The dictionary size varies from file to file and is stored in the XZ header of a file. ClamAV differs from XZ Utils when parsing this dictionary size field.

```
1  extern lzma_ret lzma_lz_decoder_init(...) {
2    ...
3    // Allocate and initialize the dictionary.
4    if (next->coder->dict.size != lz_options.dict_size) {
5      lzma_free(next->coder->dict.buf, allocator);
6      next->coder->dict.buf
7        = lzma_alloc(lz_options.dict_size, allocator);
8      ...
9
10 lzma_alloc(size_t size, const lzma_allocator *allocator) {
11   ...
12   if (allocator != NULL && allocator->alloc != NULL)
13     ptr = allocator->alloc(allocator->opaque, 1, size);
14   else
15     ptr = malloc(size);
16   ...
```

Listing 4.3: XZ Utils parses the dictionary size correctly.

As shown in Listing 4.3, XZ Utils strictly conforms to the specifications and allocates a buffer based on the permitted dictionary sizes. On the other hand, ClamAV includes an additional check on the dictionary size that deviates from the specifications. It fails to parse archives with a dictionary size greater than 182MB (line 15 in Listing 4.4). As a result of this bug, when parsing such an archive containing a malware, ClamAV does not consider the file as an archive, and thus skips scanning

the compressed malware.

```
1 SRes LzmaDec_Allocate(.., const Byte *props, ...) {
2   ...
3   dicBufSize = propNew.dicSize;
4   if (p->dic == 0 || dicBufSize != p->dicBufSize){
5     ...
6     // Invoke __xz_wrap_alloc()
7     p->dic = (Byte *)alloc->Alloc(alloc, dicBufSize);
8     if (p->dic == 0) {
9       ...
10      return SZ_ERROR_MEM;
11      ...
12
13 void *__xz_wrap_alloc(void *unused, size_t size) {
14   // Fails if size > (182*1024*1024)
15   if(!size || size > CLI_MAX_ALLOCATION)
16     return NULL;
17     ...
```

Listing 4.4: ClamAV's additional erroneous check on dictionary size.

X.509 certificate validation discrepancies

LibreSSL - Incorrect parsing of time field types: The RFC standards for X.509 certificates restrict the `Time` fields to only two forms, namely the ASN.1 representations of `UTCTime` (`YYMMDDHHMMSSZ`) and `GeneralizedTime` (`YYYYMMDDHHMMSSZ`) [78] which are 13 and 15 characters wide respectively. Time fields are also encoded with an ASN.1 tag that specifies their format. Despite the standards, in practice, we observe that 11- and 17-character time fields are used in the wild, by searching within the SSL observatory [176]. Indeed, some SSL libraries like OpenSSL and BoringSSL are more permissive while parsing such time fields.

LibreSSL, on the other hand, tries to comply strictly with the standards when parsing the validity time fields in a certificate. However, while doing so, LibreSSL introduces a bug. Unlike the other libraries, LibreSSL ignores the ASN.1 time format tag, and infers the time format type based on the length of the field (Lines 10 and

16 in Listing 4.5). In particular, the time fields in a certificate can be crafted to trick LibreSSL to erroneously parse the time fields using an incorrect type. For instance, when the time field of ASN.1 `GeneralizedTime` type is crafted to have the same length as the `UTCTime` (i.e., 13), LibreSSL treats the `GeneralizedTime` as `UTCTime`.

```
1  int asn1_time_parse(..., size_t len, ..., int mode) {
2    ...
3    int type = 0;
4    /* Constrain to valid lengths. */
5    if (len != UTCTIME_LENGTH && len != GENTIME_LENGTH)
6      return (-1);
7    ...
8    switch (len) {
9    case GENTIME_LENGTH:
10     // mode is "ignored" -- configured to 0 here
11     if (mode == V_ASN1_UTCTIME)
12       return (-1);
13     ...
14     type = V_ASN1_GENERALIZEDTIME;
15   case UTCTIME_LENGTH:
16     if (type == 0) {
17       if (mode == V_ASN1_GENERALIZEDTIME)
18         return (-1);
19       type = V_ASN1_UTCTIME;
20     }
21     ...
```

Listing 4.5: `LibreSSL` time field parsing bug.

As a result of this confusion, LibreSSL may erroneously treat a valid certificate as not yet valid, when in fact it is valid; or, it may erroneously accept an expired certificate. For example, while other libraries may interpret a `GeneralizedTime` time in *history*, `201201010101Z` as `Jan 1 01:01:00 2012 GMT`, LibreSSL will incorrectly interpret this time as a `UTCTime` time in *future*, i.e., as `Dec 1 01:01:01 2020 GMT`. Note that finding time fields of non-standard lengths in the wild suggests that CAs do not actively enforce these standards length requirement. Furthermore, we also

found certificates with `GeneralizedTime` times that are of the length 13 in the SSL observatory dataset.

GnuTLS - Incorrect validation of activation time: As shown in Listing 4.6, GnuTLS lacks a check for cases where the year is set to 0. As a result, while other SSL libraries reject a malformed certificate causing t to be 0, GnuTLS erroneously accepts it.

```
1 static unsigned int check_time_status(gnutls_x509_crt_t crt, time_t now) {
2     int status = 0;
3     time_t t = gnutls_x509_crt_get_activation_time(crt);
4     if (t == (time_t) - 1 || now < t) {
5         status |= GNUTLS_CERT_NOT_ACTIVATED;
6         status |= GNUTLS_CERT_INVALID;
7         return status;
8     ...
```

Listing 4.6: GnuTLS activation time parsing error.

BoringSSL - Incorrect representation of `KeyUsage`: According to the RFC standards, the `KeyUsage` extension defines the purpose of the certificate key and it uses a `bitstring` to represent the various uses of the key. A valid Certificate Authority (CA) certificate must have this extension present with the `keyCertSign` bit set.

BoringSSL and LibreSSL differ in the way they parse the ASN.1 `bitstring`, which is used for storing the `KeyUsage` extension in the X.509 certificates. Each `bitstring` is encoded with a "padding" byte that indicates the number of least significant unused bits in the bit representation of the structure. This byte should never be more than 7. But if the byte is set to a value greater than 7, BoringSSL fails to parse the bitstring and throws an error in Listing 4.7, whereas LibreSSL masks that byte with `0x07` and continues to parse the bitstring as-is as shown in Listing 4.8.

This subtle discrepancy results in two different interpretations of the same bitstring used in the extension. BoringSSL fails to parse the bitstring and results in an empty `KeyUsage` extension. LibreSSL, by masking the padding byte, successfully parses the extension. We also find that these libraries exhibit this discrepancy during

the parsing of a Certificate Signing Request (CSR). This can have critical security implications. Consider the scenario where a CA using BoringSSL parses such a CSR presented by an attacker and does not interpret the extension correctly. The CA misinterprets the key usages and does not detect certain blacklisted ones. In this situation, the CA might copy the malformed extension to the issued certificate. Subsequently, when the issued certificate is parsed by a client using LibreSSL, it will be parsed with a valid keyUsage extension and thus the attacker can use the certificate for purposes that were not intended by the CA.

```
1 ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp) {
2   ...
3   p = *pp;
4   padding = *(p++);
5   // returns an error if invalid padding byte
6   if (padding > 7) {
7     OPENSSL_PUT_ERROR(ASN1, ASN1_R_INVALID_BIT_STRING_BITS_LEFT);
8     goto err;
9   }
10  ret->flags &= ~(ASN1_STRING_FLAG_BITS_LEFT | 0x07);
11  ret->flags |= (ASN1_STRING_FLAG_BITS_LEFT  | i);
12  ...
```

Listing 4.7: BoringSSL code for validating `bitstrings`.

```
1 ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp) {
2   ...
3   p = *pp;
4   i = *(p++);
5   // masks the padding byte, instead of with a check
6   ret->flags&= ~(ASN1_STRING_FLAG_BITS_LEFT| 0x07);
7   ret->flags|=(ASN1_STRING_FLAG_BITS_LEFT  | (i&0x07));
8   ...
```

Listing 4.8: LibreSSL code for validating `bitstrings`.

PDF viewer discrepancies: As mentioned in Section 4.2.4.2, NEZHA uncovered 7 unique discrepancies in the tested three PDF browsers (Evince, Xpdf and MuPDF)

over a total of 10 million generations. Examples of the found discrepancies include PDF files that could be opened in one viewer but not another and PDFs rendered with different contents across viewers. One interesting discrepancy includes a PDF that Evince treats as encrypted (thus opening it with a password prompt) but Xpdf recognizes as unencrypted (MuPDF and Xpdf abort with errors trying to render the file).

4.2.4.7 Memory Corruption Bugs

In addition to finding semantic bugs, by leveraging Clang's sanitization passes, Nezha was equally successful in uncovering memory corruption vulnerabilities and crashes. In particular, during our experiments Nezha uncovered a total of 8 memory errors, 5 of which were memory corruption bugs.

ClamAV use-after-free: Nezha disclosed a use-after-free heap bug in ClamAV, which is invoked when parsing a malformed XZ archive. Before scanning a XZ archive, ClamAV first copies it in memory into a single memory buffer. To do so, it allocates and deallocates memory dynamically. However, every time this buffer is freed, its address is not subsequently set to `NULL`. This coding error consequently leads to a use-after-free bug in the cases where multiple blocks are present in the XZ archive. Despite the fact that the buffer is not set to `NULL` when freed, the ClamAV library only allocates new memory for the buffer if its address is not `NULL`. Thus, failure to set the buffer address to `NULL` in an earlier `free` operation makes the library erroneously skip the allocation overall. An attacker can exploit this vulnerability by sending a malformed XZ archive that will crash ClamAV when ClamAV attempts to scan the archive. In order for the crash to occur, the following conditions need to hold: i) the archive must have at least two blocks, of which the number of filters in two consecutive blocks need to be different and ii) the first block in the archive needs to have at least one filter.

wolfSSL memory errors: NEZHA uncovered four memory corruption bugs in wolfSSL, all of which were marked as critical by the wolfSSL developers and patched within six days after we reported the bugs. Two of the bugs were caused by missing checks for malformed PEM certificate headers inside the `PemToDer` function, which converts a X.509 certificate from PEM to DER format. The missing checks resulted in out-of-bounds memory reads. The third bug was caused by a missing check for the return value of a `PemToDer` call, inside the `wolfSSL_CertManagerVerifyBuffer` routine, causing a segmentation fault. In this case, the structure holding the DER-converted certificate is corrupted. Finally the fourth bug, also occurring inside `Pem2Der`, resulted in an out-of-bounds read, due to a missing check on the size of the PEM certificate to be converted. This can be triggered by an intermediate certificate in a chain that has the correct PEM header but an empty body: the missing check will cause `Pem2Der` to not return any error, which in turn results in an out-of-bounds memory access during the subsequent steps of the verification process.

GnuTLS null pointer dereference: NEZHA found a missing check inside the `gnutls_oid_to_ecc_curve` routine of GnuTLS, where dereferenced pointers were not checked to be not `NULL`. This bug resulted in a segmentation fault while parsing an appropriately crafted certificate.

4.3 Discussion

In this Chapter, we presented differential diversity (δ-diversity), a novel metric to be used for selective, context-aware guidance, and also presented and evaluated NEZHA, the first, to the best of our knowledge, generic differential fuzzer targeting both crash-inducing and semantic bugs. We demonstrated how traditional coverage-based testing tools can be augmented to support context-aware analyses, utilizing compile-assisted instrumentation under δ-diversity guidance. Although being context-aware with re-

spect to what portions of the code are executed, the compile-time instrumentation utilized by NEZHA is still agnostic to the potential similarities in the binaries of the tested applications. This is because NEZHA's engine will give a high score to inputs that explore new functions across the tested applications, despite the fact that these functions might be identical. Future work could address this limitation by ignoring, during the scoring of different inputs across binaries, functions that identified to be identical or of similar functionality across binaries [6, 188, 41, 187]. Thus, an input will be deemed interesting only if it explores paths that have *semantic diversity* across the tested binaries, and not simply path diversity. To this end, NEZHA's engine could build upon existing work on detection of binary similarities [59, 170, 188, 41, 187, 57, 109, 56] so as to obtain a similarity score for the functions and CFG parts of the applications that are differentially tested, utilizing context-aware diversity metrics that build upon the aforementioned semantic differences.

Chapter 5

Evolutionary Testing for Complexity Vulnerabilities

In the previous Chapter, we demonstrated how adopting context-aware guidance may extend the scope of existing toolchains so that the latter may target broader classes of errors. To this end, we described Nezha, which augmented state-of-the-art evolutionary fuzzing engines to target semantic bugs, additionally to crash-inducing vulnerabilities. To further demonstrate the agility of context-aware guidance, in this Chapter, we will retrofit *the same* state-of-the-art fuzzer architecture, however to now target algorithmic complexity vulnerabilities instead of logic bugs. To this end, we will present and evaluate SlowFuzz, the first, to the best of our knowledge, generic evolutionary fuzzer targeting complexity vulnerabilities. Although Nezha and SlowFuzz are presented and evaluated, for the sake of clarity, separately, they share the same context-agnostic components, and can serve as different facets of a single, *adaptive* fuzzer, capable of targeting different types of errors based on the analyst's preferences.

5.1 Background

Algorithmic complexity vulnerabilities occur when the worst-case time/space complexity of an application is significantly higher than the respective average case for particular user-controlled inputs. When such conditions are met, an attacker can launch Denial-of-Service attacks against a vulnerable application by providing inputs that trigger this worst-case behavior, or force the system to a state where resources are under-utilized.

Such attacks have repeatedly been encountered in the wild, causing serious effects on production systems, taking down entire websites [166], disabling/bypassing Web Application Firewalls (WAF) [46], or keeping thousands of CPUs busy by merely performing hash-table insertions [138, 196]. Crosby et al. [45] were the first to present complexity attacks abusing collisions in hash table implementations and, since then, several lines of work have explored a variety attacks related to complexity vulnerabilities: Cai et al. [30] leveraged complexity vulnerabilities in the Linux kernel name lookup hash tables to exploit race conditions in the kernel `access(2)/open(2)` system calls, whereas Sun et al. [171] explored complexity vulnerabilities in the name lookup algorithm of the Linux kernel to achieve an exploitable covert timing channel. Smith et al. [163] exploited the syntax of the Snort IDS to perform a complexity attack resulting in slowdowns during packet inspection, whereas Shenoy et al. [158, 157] presented an algorithmic complexity attack against the popular Aho-Corasick string searching algorithm.[1]

Unfortunately, detection of algorithmic complexity vulnerabilities in a domain-independent way in practice is a hard, multi-faceted problem. It is often infeasible to completely abandon algorithms or data structures with high worst-case complexities without severely restricting the functionality or backwards-compatibility of an appli-

[1] And also proposed hardware and software-based defenses to mitigate the worst-case performance of their attacks.

cation. Moreover, manual time complexity analysis of real-world applications is hard to scale, whereas asymptotic complexity analysis ignores the constant factors that can significantly affect the application execution time despite not impacting the overall complexity class. All these factors significantly harden the detection of algorithmic complexity vulnerabilities.

Even when real-world applications use well-understood algorithms, time complexity analysis is still non-trivial for the following reasons. First, the time/space complexity analysis changes significantly even with minor implementation variations (for instance, the choice of the pivot in the quicksort algorithm drastically affects its worst-case runtime behavior [42]). Reasoning about the effects of such changes requires significant manual effort. Second, most real-world applications often have multiple inter-connected components that interact in complex ways. This interconnection further complicates the estimation of the overall complexity, even when the time complexity of the individual components is well understood.

Most existing detection mechanisms for algorithmic complexity vulnerabilities use domain- and implementation-specific heuristics or rules, e.g., especially focusing on backtracking during the matching process [198, 15, 124, 91]. However, such rules tend to be brittle and are hard to scale to a large number of diverse domains, since their creation and maintenance requires significant manual effort and expertise. Moreover, keeping such rules up-to-date with newer software versions is onerous, as even minor changes to the implementation might require significant changes in the rules.

In this Chapter, we will demonstrate how to augment existing fuzzing infrastructures to target algorithmic complexity errors. To do so, we will construct a guidance engine that favors inputs that cause large variations in resource utilization in the tested application, measured through the number of executed instructions or CPU usage. We assume that our tool has gray-box access to the application binary, i.e., it can instrument the binary in order to harvest different fine-grained resource usage

information from multiple runs of the binary, with different inputs. Note that our goal is not to estimate the asymptotic complexities of the underlying algorithms or data structures of the application. Instead, we measure the resource usage variation in some pre-defined metric like the total edges accessed during a run, and try to maximize that metric. Even though, in most cases, the inputs causing worst-case behaviors under such metrics will be the ones demonstrating the actual worst-case *asymptotic* behaviors, but this may not always be true due to the constant factors ignored in the asymptotic time complexity, the small input sizes, etc.

In our threat model, we assume an attacker can provide arbitrary specially-crafted inputs to the vulnerable software to trigger worst-case behaviors. Such a model is consistent with production-level systems as most non-trivial real-world software like Web applications and regular expression matchers need to deal with inputs from untrusted sources. For a subset of our experiments involving regular expression matching, we assume that attackers can control regular expressions provided to the matchers. This is a valid assumption for a large set of applications that provide search functionality through custom regular expressions from untrusted users.

5.2 A Motivating Example

In order to understand how feedback-driven fuzzers can provide context-aware guidance to focus on complexity vulnerabilities, let us consider quicksort, one of the simplest yet most widely used sorting algorithms. It is well-known [42] that quicksort has an average time complexity of $O(n \log n)$ but a worst-case complexity of $O(n^2)$ where n is the size of the input. However, finding an actual input that demonstrates the worst-case behavior in a particular quicksort implementation depends on low-level details like the pivot selection mechanism. If an adversary knows the actual pivot selection scheme used by the implementation, she can use domain-specific rules

to find an input that will trigger the worst-case behavior (e.g., the quadratic time complexity) [115].

```
function quicksort(array):
    /* initialize three arrays to hold
    elements smaller, equal and greater
    than the pivot */
    smaller, equal, greater = [], [], []
    if len(array) <= 1:
        return
    pivot = array[0]
    for x in array:
        if x > pivot:
            greater.append(x)
        else if x == pivot:
            equal.append(x)
        else if x < pivot:
            smaller.append(x)
    quicksort(greater)
    quicksort(smaller)
    array = concat(smaller, equal, greater)
```

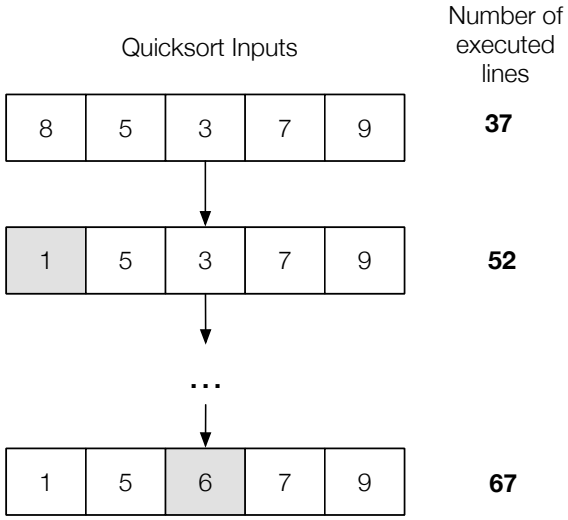

Figure 5.1: Pseudocode for quicksort with a simple pivot selection mechanism and overview of SlowFuzz's evolutionary search process for finding inputs that demonstrate worst-case quadratic time complexity. The shaded boxes indicate mutated inputs.

However, in our setting, SlowFuzz does not know any domain-specific rules. It also does not understand the semantics of pivot selection or which part of the code implements the pivot selection logic, even though it has access to the quicksort implementation. We would still like SlowFuzz to generate inputs that trigger the

corresponding worst-case behavior and identify the algorithmic complexity vulnerability.

This brings us to the following question: *how can* SlowFuzz *automatically generate inputs that would trigger worst-case performance in a tested binary in a domain-independent manner?* The search space of all inputs is too large to search exhaustively. Our key intuition is that evolutionary search techniques can be used to iteratively find inputs that are closer to triggering the worst-case behavior, if they are properly augmented Adopting an evolutionary testing approach, SlowFuzz begins with a corpus of seed inputs, applies mutations to each of the inputs in the corpus, and ranks each of the inputs based on their resource usage patterns. SlowFuzz keeps the highest ranked inputs for further mutations in upcoming generations.

To further illustrate this point, let us consider the pseudocode of Figure 5.1, depicting a quicksort example with a simple pivot selection scheme—the first element of the array being selected as the pivot. In this case, the worst-case behavior can be elicited by an already sorted array. Let us also assume that SlowFuzz's initial corpus consists of some arrays of numbers and that none of them are completely sorted. Executing this quicksort implementation with the seed arrays will result in a different number of statements/instructions executed based on how close each of these arrays are to being sorted. SlowFuzz will assign a score to each of these inputs based on the number of statements executed by the quicksort implementation for each of the inputs. The inputs resulting in the highest number of executed statements will be selected for further mutation to create the next generation of inputs. Therefore, each upcoming generation will have inputs that are closer to being completely sorted than the inputs of the previous generations.

For example, let us assume the initial corpus for SlowFuzz consists of a single array $\mathcal{I} = [8, 5, 3, 7, 9]$. At each step, SlowFuzz picks at random an input from the corpus, mutates it, and passes the mutated input to the above quicksort implementa-

tion while recording the number of executed statements. As shown in Figure 5.1, the input [8, 5, 3, 7, 9] results in the execution of 37 lines of code (LOC). Let us assume that this input is mutated into $[1, 5, 3, 7, 9]$ that causes the execution of 52 LOC which is higher than the original input and therefore [1, 5, 3, 7, 9] is selected for further mutation. Eventually, SlowFuzz will find a completely sorted array (e.g., [1, 5, 6, 7, 9] as shown in Figure 5.1) that will demonstrate the worst-case quadratic behavior. We provide a more thorough analysis of SlowFuzz's performance on various sorting implementations in Section 5.3.3.2.

5.3 SlowFuzz

5.3.1 Methodology

The key observation for our methodology is that evolutionary search techniques together with dynamic analysis present a promising approach for finding inputs that demonstrate worst-case complexity of a test application in a domain-independent way. However, to enable SlowFuzz to efficiently find such inputs, we need to carefully design effective guidance mechanisms and mutation schemes to drive SlowFuzz's input generation process. We design a new evolutionary algorithm with customized guidance mechanisms and mutation schemes that are tailored for finding inputs causing worst-case behavior.

Algorithm 4 shows the core evolutionary engine of SlowFuzz. Initially, SlowFuzz randomly selects an input to execute from a given seed corpus (line 4), which is mutated (line 5) and passed as input to the test application (line 6). During an execution, profiling info such as the different types of resource usage of the application are recorded (lines 6-8). An input is scored based on its resource usage and is added to the mutation corpus if the input is deemed as a slow unit (lines 9-12).

In the following Sections, we describe the core components of SlowFuzz's engine,

Algorithm 4 SlowFuzz: Report all slow units for application $\mathcal{A}$ after n generations, starting from a corpus $\mathcal{I}$

```
1:  procedure DiffTest(I, A, n, GlobalState)
2:      units = ∅ ;reported slowunits
3:      while generation ≤ n and I ≠ ∅ do
4:          input = RandomChoice(I)
5:          mut_input = Mutate(input)
6:          app_insn, app_outputs = Run(A, mut_input)
7:          gen_insn ∪ = {app_insn}
8:          gen_usage ∪ = {app_usage}
9:          if SlowUnit(gen_insn, gen_usage,
                        GlobalState) then
10:             I ← I ∪ mut_input
11:             units ∪ = mut_input
12:         end if
13:         generation = generation + 1
14:     end while
15:     return units
16: end procedure
```

particularly the fitness function used to determine whether an input is a slow unit or not, and the offset and type of mutations performed on each of the individual inputs in the corpus.

5.3.1.1 Fitness Functions

As shown in Algorithm 4, SlowFuzz determines, after each execution, whether the executed unit should be considered for further mutations (lines 9-12). SlowFuzz ranks the current inputs based on the scores assigned to them by a fitness function and keeps the fittest ones for further mutation. Popular coverage-based fitness functions which are often used by evolutionary fuzzers to detect crashes, are not well suited for our purpose as they do not consider loop iterations which are crucial for detecting worst-case time complexity.

SlowFuzz's input generation is guided by a fitness function based on resource usage. Such a fitness function is generic and can take into consideration different kinds of resource usage like CPU usage, energy, memory, etc. In order to measure

the CPU usage in a fine-grained way, SlowFuzz's fitness function keeps track of the total count of all instructions executed during a run of a test program. The intuition is that the test program becomes slower as the number of executed instructions increases. Therefore, the fitness function selects the inputs that result in the highest number of executed instructions as the slowest units. For efficiency, we monitor execution at the basic-block level instead of instructions while counting the total number of executed instructions for a program. We found that this method is more effective at guiding input generation than directly using the time taken by the test program to run. The runtime of a program shows large variations, depending on the application's concurrency characteristics or other programs that are executing in the same CPU, and therefore is not a reliable indicator for small increases in CPU usage. Similarly to the fitness functions of Nezha, context-aware guidance can be extended to multiple programs using the appropriate δ-diversity metrics. In Section 5.3.1.2, we will examine how we can achieve context-aware prioritization in different components of the fuzzer engine, such as in the modules responsible for the mutation operations.

5.3.1.2 Mutation Strategies

Traditionally, feedback-driven mutational fuzzers support a series of different mutation operations, and randomly select the mutation to perform at each stage of the input generation. SlowFuzz builds on top of libFuzzer, however implements a series of novel mutation strategies that favor locality, to more effectively trigger potential complexity vulnerabilities. Essentially, instead of blindly (i.e., in a context-agnostic manner) selecting the mutation to be performed at each step, SlowFuzz introduces several context-aware mutation strategies, based on partitions of both the inputs as well as of the different mutation operators. These strategies decide which mutation operations to apply and which byte offsets in an input to modify, to generate a new mutated input (Algorithm 4, line 5).

Regardless of the strategy in place, SlowFuzz supports the following mutation *operations*: (i) add/remove a new/existing byte from the input; ii) randomly modify a bit/byte in the input; iii) randomly change the order of a subset of the input bytes; iv) randomly change bytes whose values are within the range of ASCII codes for digits (i.e., 0x30-0x39); v) perform a crossover operation in a given buffer mixing different parts of the input; and vi) mutate bytes solely using characters or strings from a user-provided dictionary. In the following Section, we describe the strategies supported by SlowFuzz. Section 5.3.3.6 presents a performance comparison of these strategies.

Random Mutations: Random mutations are the simplest mutation strategy supported by SlowFuzz. Under this mutation strategy, one of the aforementioned mutations is selected at random and is applied on an input, as long as it does not violate other constraints for the given testing session, such as exceeding the maximum input length specified by the auditor. This strategy is similar to the ones used by popular evolutionary fuzzers like AFL [206] and libFuzzer [105] for finding crashes or memory safety issues.

Mutation Priority: Under this strategy, the mutation operation is selected with ϵ probability based on its success at producing slow units during previous executions. The mutation operation is picked at random with $(1-\epsilon)$ probability. In contrast, the mutation offset is still selected at random just like the strategy described above.

In particular, during testing, we count all the cases in which a mutation operation resulted in an increase in the observed instruction count and the number of times that operation has been selected. Based on these values, we assign a score to each mutation operation denoting the probability of the mutation to be successful at increasing the instruction count. For example, a score of 0 denotes that the mutation operation has never resulted in an increase in the number of executed instructions, whereas a score of 1 denotes that the mutation always resulted in an increase.

We pick the highest-scoring mutation among all mutation operations with a prob-

ability ϵ. The tunable parameter ϵ determines how often a mutation operation will be selected at random versus based on its score. Essentially, different values of ϵ provide different trade-offs between exploration and exploitation. In SlowFuzz, we set the default value of ϵ to 0.5.

Offset Priority: This strategy selects the mutation operation to be applied randomly at each step, but the offset to be mutated is selected based on prior history of success at increasing the number of executed instructions. The mutation offset is selected based on the results of previous executions with a probability ϵ and at random with a probability $(1 - \epsilon)$. In the first case, we select the offset that showed the most promise based on previous executions (each offset is given a score ranging from 0 to 1 denoting the percentage of times in which the mutation of that offset led to an increase in the number of instructions).

Hybrid: In this last mode of operation we apply a combination of both mutation and offset priority as described above. For each offset, we maintain an array of probabilities of success for each of the mutation operations that are being performed. Instead of maintaining a coarse-grained success probability for each mutation in the mutation priority strategy, we maintain fine-grained success probabilities for each offset/mutation operation pairs. We compute the score of each offset by computing the average of success probabilities of all mutation operations at that offset. During each mutation, with a probability of ϵ, we pick the offset and operation with the highest scores. The mutation offset and operation are also picked randomly with a probability of $(1 - \epsilon)$.

5.3.2 Implementation

The SlowFuzz prototype is built atop of libFuzzer [105], a popular evolutionary fuzzer for finding crash and memory safety bugs. We outline the implementation details of different components of SlowFuzz below. Overall, our modifications to

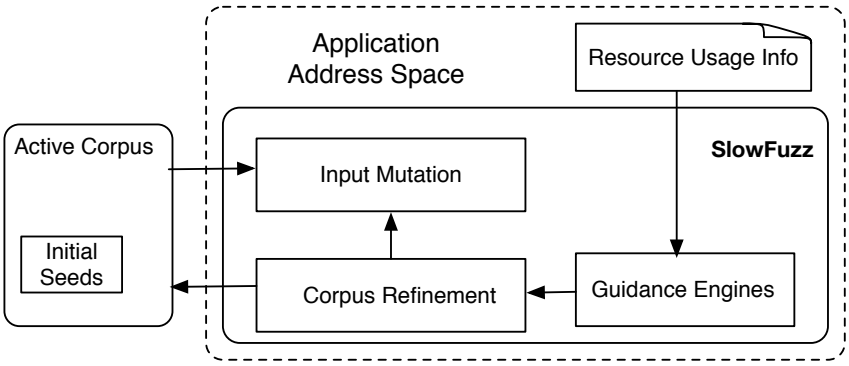

Figure 5.2: SlowFuzz Architecture.

libFuzzer consist of 550 lines of C++ code. We used Clang v4.0 for compiling our modifications along with the rest of libFuzzer code.

Figure 5.2 shows SlowFuzz's high-level architecture. Similar to the popular evolutionary fuzzers like AFL [206] and libFuzzer [105], SlowFuzz executes in the same address space as the application being tested. We instrument the test application so that SlowFuzz can have access to different resource usage metrics (e.g, number of instructions executed) needed for its analysis. The instrumented test application subsequently is executed under the control of SlowFuzz's analysis engine. SlowFuzz maintains an active corpus of inputs to be passed into the tested applications and refines the corpus during execution based on SlowFuzz's fitness function. For each generation, an input is selected, mutated, then passed into the `main` routine of the application for its execution.

Instrumentation: Similar to libFuzzer, SlowFuzz's instrumentation is based on Clang's SanitizerCoverage [152] passes. Particularly, SanitizerCoverage allows tracking of each executed function, basic block, and edge in the Control Flow Graph (CFG). It also allows us to register callbacks for each of these events. SlowFuzz makes use of SanitizerCoverage's eight bit counter capability that maps each Control Flow Graph (CFG) edge into an eight bit counter representing the number of times that edge was accessed during an execution. We use the counter to keep track of the

following ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+. This provides a balance between accuracy of the counts and the overhead incurred for maintaining them. This information is then passed into SlowFuzz's fitness function, which determines whether an input is slow enough to keep for the next generation of mutations.

Mutations: LibFuzzer provides API support for custom input mutations. However, in order to implement the mutation strategies proposed we had to modify libFuzzer internals. Particularly, we augment the functions used in libFuzzer's `Mutator` class to return information on the mutation operation, offset, and the range of affected bytes for each new input generated by LibFuzzer. This information is used to compute the scores necessary for supporting mutation piority, offset priority, and hybrid

5.3.3 Evaluation

In this Section, we evaluate SlowFuzz on the following objectives: a) Is SlowFuzz capable of generating inputs that match the theoretical worst-case complexity for a given algorithm's implementation? b) Is SlowFuzz capable of efficiently finding inputs that cause performance slowdowns in real-world applications? c) How do the different mutation and guidance engines of SlowFuzz affect its performance? d) How does SlowFuzz compare with code-coverage-guided search at finding inputs demonstrating worst-case application behavior?

We describe the detailed results of our evaluation in the following Sections. All our experiments were performed on a machine with 23GB of RAM, equipped with an Intel(R) Xeon(R) CPU X5550 @ 2.67GHz and running 64-bit Debian 8 (jessie), compiled with GCC version 4.9.2, with a kernel version 4.5.0. All binaries were compiled using the Clang-4.0 compiler toolchain. All instruction counts and execution times are measured using the Linux `perf` profiler v3.16, averaging over 10 repetitions for each `perf` execution.

5.3.3.1 Overview

In order to adequately address the questions outlined in the previous Section, we execute SlowFuzz on applications of different algorithmic profiles and evaluate its ability of generating inputs that demonstrate worst case behavior.

First, we examine if SlowFuzz generates inputs that demonstrate the theoretical worst-case behavior of well-known algorithms. We apply SlowFuzz on sorting algorithms with well-known complexities. The results are presented in Section 5.3.3.2. Subsequently, we apply SlowFuzz on different applications and algorithms that have been known to be vulnerable to complexity attacks: the PCRE regular expression library, the default hash table implementation of PHP, and the `bzip2` binary. In all cases, we demonstrate that SlowFuzz is able to trigger complexity vulnerabilities. Table 5.1 shows a summary of our findings.

Table 5.1: Result Summary for Applications Tested With SlowFuzz.

Tested Application	**Fuzzing Outcome**
Insertion sort [42]	41.59x slowdown
Quicksort (Fig 5.1)	5.12x slowdown
Apple quicksort	3.34x slowdown
OpenBSD quicksort	3.30x slowdown
NetBSD quicksort	8.7% slowdown
GNU quicksort	26.36% slowdown
PCRE (fixed input)	78 exponential & 765 superlinear regexes
PCRE (fixed regex)	8% - 25% slowdown
PHP hashtable	20 collisions in 64 keys
`bzip2` decompression	~300x slowdown

As shown in Table 5.1, SlowFuzz is successful at inducing significant slowdown on all tested applications. Moreover, when applied to the PCRE library, it managed to generate regular expressions that exhibit exponential and super-linear (worse than quadratic) matching automatically, without any knowledge of the structure of a regular expression. Likewise, it successfully generated inputs that induce a high number of collisions when inserted into a PHP hash table, without any notion of

hash functions. In the following Sections, we provide details on each of the above test settings.

5.3.3.2 Sorting

Simple quicksort and insertion sort: Our first evaluation of SlowFuzz's consistency with theoretical results is performed on common sorting algorithms with well-known worst-performing inputs. To this end, we initially apply SlowFuzz on an implementation of the insertion sort algorithm [42], as well as on an implementation of quicksort [42] in which the first sub-array element is always selected as the pivot. Both of the above implementations demonstrate quadratic complexity when the input passed to them is sorted. We run SlowFuzz for 1 million generations on the above implementations, sorting a file with a size of 64 bytes, and examine the slowdown SlowFuzz introduced over the fastest unit seen during testing. To do so, we count the total instructions executed by each program for each of the inputs, subtracting all instructions not relevant to the quicksort functionality (e.g., loader code). Our results are presented in Figure 5.3.

Figure 5.3 represents an average of 100 runs. In each run, SlowFuzz started execution with a single random 64 byte seed, and executed for 1 million generations. We notice that SlowFuzz achieves 41.59x and 5.12x slowdowns for insertion sort and quicksort respectively. In order to examine how this behavior compares to random testing, we randomly generated 1 million inputs of 64 bytes each and measured the instructions required for insertion sort and quicksort, respectively. Figure 5.3 depicts the *maximum* slowdown achieved through random testing *across all* runs. We notice that in both cases SlowFuzz outperforms the brute-force worst-input estimation. Finally, we observe that the gap between brute-force search and SlowFuzz is much higher for quicksort than insertion, which is consistent with the fact that average case complexity of insertion sort is $O(n^2)$, compared to quicksort's $O(n \log n)$. Therefore,

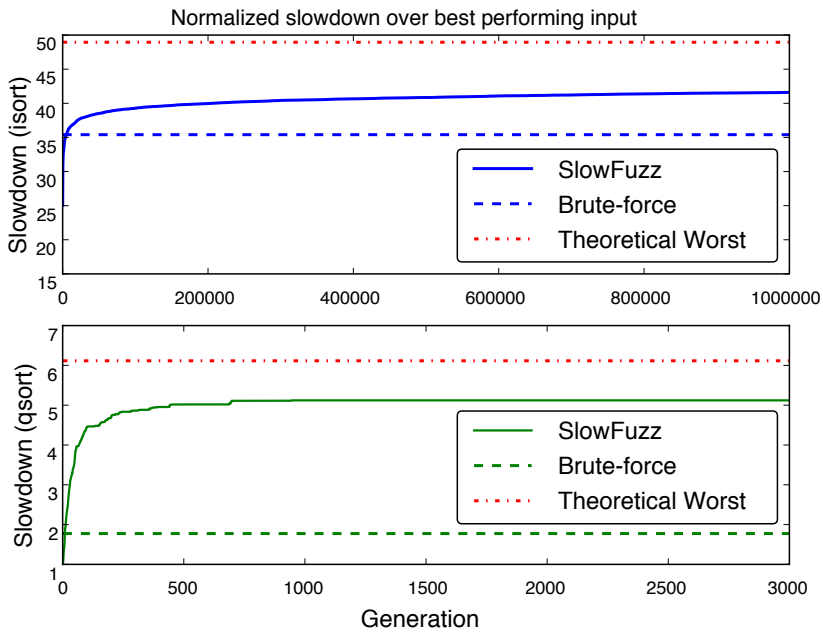

Figure 5.3: Best slowdown achieved by SlowFuzz at each generation (normalized over the slowdown of the best-performing input) versus best random testing outcome, on our insertion sort and quicksort drivers, for an input size of 64 bytes (average of 100 runs). The SlowFuzz achieves slowdowns of 84.97% and 83.74% compared to the theoretical worst cases for insertion sort and quicksort respectively.

a random input is more likely to demonstrate worst-case behavior for insertion sort but not for quicksort.

Real-world quicksort implementations: We also examined how SlowFuzz performs when applied to real-world quicksort implementations. Particularly, we applied it to the Apple [76], GNU [64], NetBSD [125], and OpenBSD [104] quicksort implementations. We notice that SlowFuzz's performance on real world implementations is consistent with the quicksort performance that we observed in the experiments described above. In particular, the slowdowns generated by SlowFuzz were (in increasing order) 8.7%, for theNetBSD implementation, 26.36% for the GNU quicksort implementation, 3.30x for the OpenBSD implementation and 3.34x for the Apple implementation. We notice that, despite the fact these implementations use efficient pivot selection strategies, SlowFuzz still manages to trigger significant slowdowns. On the contrary, repeating the same experiment using naive coverage-based fuzzing

yields slowdowns that never surpass 5% for any of the libraries. This is an expected result, as coverage-based fuzzers are geared towards maximizing coverage, and thus do not favor inputs exercising the same edges repeatedly over inputs that discover new edges.

Finally, we note that, similar to the experiment of Figure 5.3, the slowdowns for Figure 5.4 are also measured in terms of executed instructions, normalized over the instructions of the best performing input seen during testing.

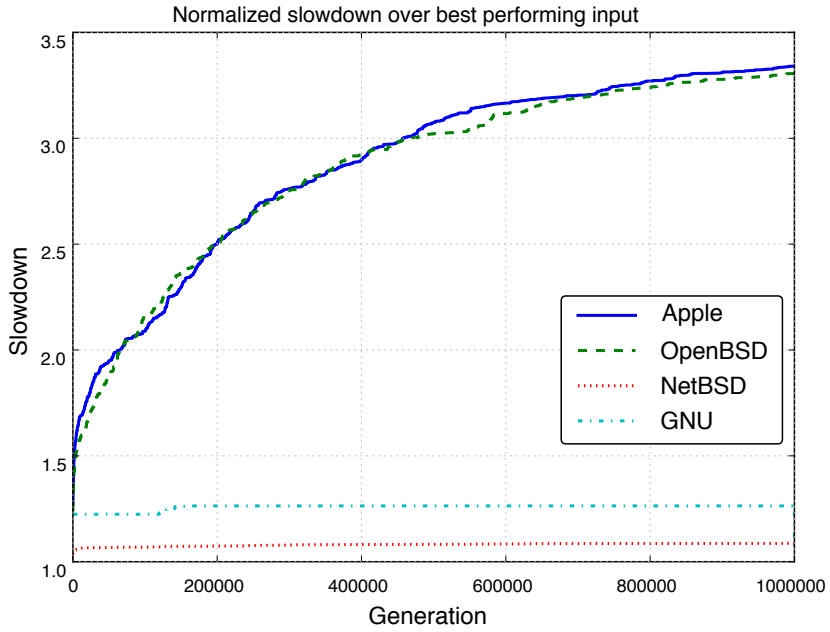

Figure 5.4: Best slowdown (with respect to the best-performing input) achieved by SlowFuzz at each generation normalized over the best random testing outcome, on real-world quicksort implementations, for an input size of 64 bytes (average of 100 runs).

5.3.3.3 Regular Expressions

Regular expression implementations are known to be susceptible to complexity attacks [196, 147, 129]. In particular, there are over 150 Regular expression Denial of Service (ReDoS) vulnerabilities registered in the National Vulnerability Database (NVD), which are the result of exponential (e.g., [48]) or super-linear (worse than

quadratic) e.g., [47] complexity of regular expression matching by several existing matchers [198].

Even performing domain-specific analyses of whether an application is susceptible to ReDoS attacks is non-trivial. Several works are solely dedicated to the detection of exploitation of such vulnerabilities. Recently, Rexploiter [198] presented algorithms to detect whether a given regular expression may result in non-deterministic finite automata (NFA) that require super-linear or exponential matching times for specially crafted inputs. They have also presented domain-specific algorithms to generate inputs capable of triggering such worst-case performance. The above denote the hardness of SlowFuzz's task, namely finding regular expressions that may result in super-linear or exponential matching times without any domain knowledge.

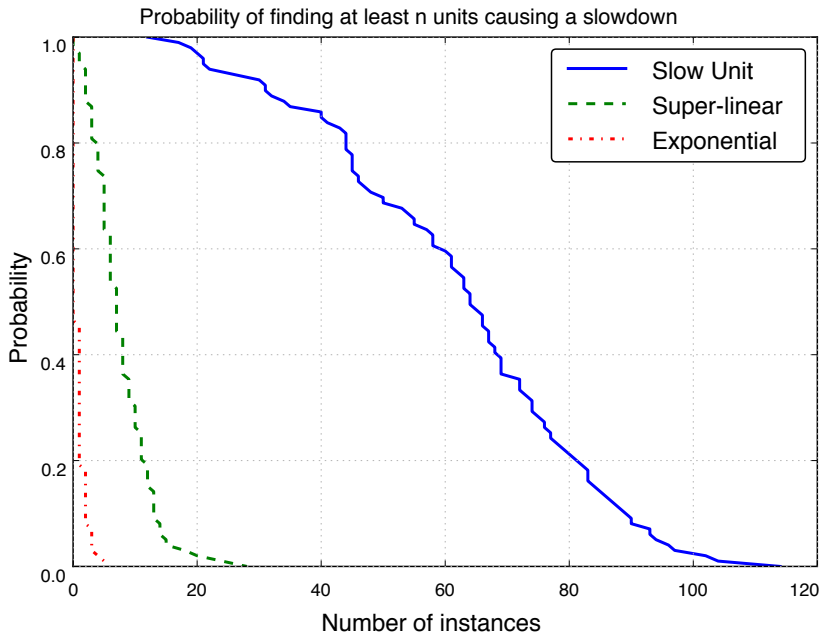

Figure 5.5: Probability of SlowFuzz finding at least n unique instances of regexes that cause a slowdown, or exhibit super-linear and exponential matching times, after 1 million generations (inverse CDF over 100 runs).

For the regular expression setting we perform two separate experiments to check whether SlowFuzz can produce i) regular expressions which exhibit super-linear and exponential matching times, ii) inputs that cause slowdown during matching,

given a fixed regular expression. To this end, we apply SlowFuzz on the PCRE regular expression library [132] and provide it with a character set of the symbols used in PCRE-compliant regular expressions (in the form of a dictionary). Notice that we do not further guide SlowFuzz with respect to what mutations should be done and SlowFuzz's engine is completely agnostic of the structure of a valid regular expression. In all cases, we start testing from an empty corpus without providing any seeds of regular expressions to SlowFuzz.

Fixed string and mutated regular expressions: For the first part of our evaluation, we apply SlowFuzz on a binary that utilizes the PCRE library to perform regular expression matching and we let SlowFuzz mutate the regular expression part of the `pcre2_match` call used for the matching, using a dictionary of regular expression characters. The input to be matched against the regular expression is selected from a random pool of characters and SlowFuzz executes for a total of 1 million generations, or until a time-out is hit. The regular expressions generated by SlowFuzz are kept limited to 10 characters or less. Once a SlowFuzz session ends, we evaluate the time complexity of the generated regular expressions utilizing `Rexploiter` [198], which detects if the regular expression is super-linear, exponential, or none of the two. We repeat the above process for a total of 100 fuzzing sessions.

Overall, SlowFuzz generates a total of 33343 regular expressions during the above 100 sessions, out of which 27142 are rejected as invalid whereas 6201 are valid regular expressions that caused a slowdown. Out of the valid regular expressions, 765 are superlinear and 78 are exponential. This experiment demonstrates that despite being agnostic of the semantics of regex matching, SlowFuzz successfully generates regexes requiring super-linear and exponential matching times. Six such examples are presented in Table 5.2.

A detailed case study: The regexes presented in Table 5.2 are typical examples of regular expressions that require non-linear matching running times. This happens

Table 5.2: Sample regexes generated by SlowFuzz resulting in super-linear (greater than quadratic) and exponential matching complexity.

Super-linear (greater than quadratic)	Exponential
c*ca*b*a*b	(b+)+c
a+b+b+b+a+	c*(b+b)+c
c*c+ccbc+	a(a|a*)+a

due to the existence of different paths in the respective NFAs, which reach the same state through an identical sequence of labels. Such paths have a devastating effect during backtracking [198]. To further elaborate on this property, let us consider the NFA depicted in Figure 5.6, which corresponds to the regular expression (b+)+c of Table 5.2.

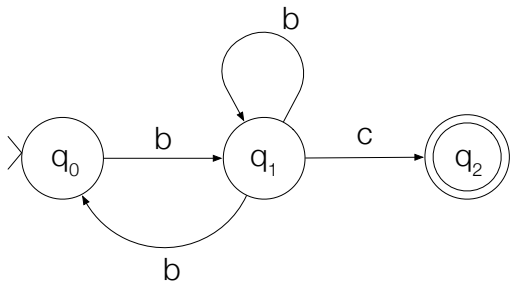

Figure 5.6: NFA for the regular expression (b+)+c suffering from exponential matching complexity as found by SlowFuzz. q_0 is the entry state, q_2 the accept state, and q_1 the pivot state for the exponential backtracking.

We notice that, for the NFA shown in Figure 5.6, starting from state q_1, it is possible to reach q_1 again, through two different paths, namely the paths ($q_1 \xrightarrow{b} q_0, q_0 \xrightarrow{b} q_1$) and ($q_1 \xrightarrow{b} q_1, q_1 \xrightarrow{b} q_1$). Moreover, we notice that the labels in the transitions for both of the above paths are the same: 'bb' is consumed in both cases. Thus, as it is possible to reach q_2 from q_1 (via label c) as well as reach q_1 from the initial state q_0, there will be an exponentially large number of paths to consider in the case of backtracking. Similar issues arise with loops appearing in NFAs with super-linear matching [198].

As mentioned above, on average, among the valid regular expressions generated by SlowFuzz, approximately 12.33% of the regexes have super-linear matching com-

plexity, whereas 2.29% on average have exponential matching complexity. The aforementioned results are aggregates across all the 100 executions of the experiment. In order to estimate the probability of SlowFuzz to generate a regex that exhibits a slowdown,[2] or super-linear and exponential matching times in a *single* session, we calculate the respective inverse CDF which is shown in Figure 5.5. We notice that, for all the regular expressions observed, SlowFuzz successfully generates inputs that incur a slowdown during matching. In particular, with 90% probability, SlowFuzz generates at least 2 regular expressions requiring super-linear matching time and at least 31 regular expressions that cause a slowdown. SlowFuzz generates at least one regex requiring exponential matching time with a probability of 45.45% .

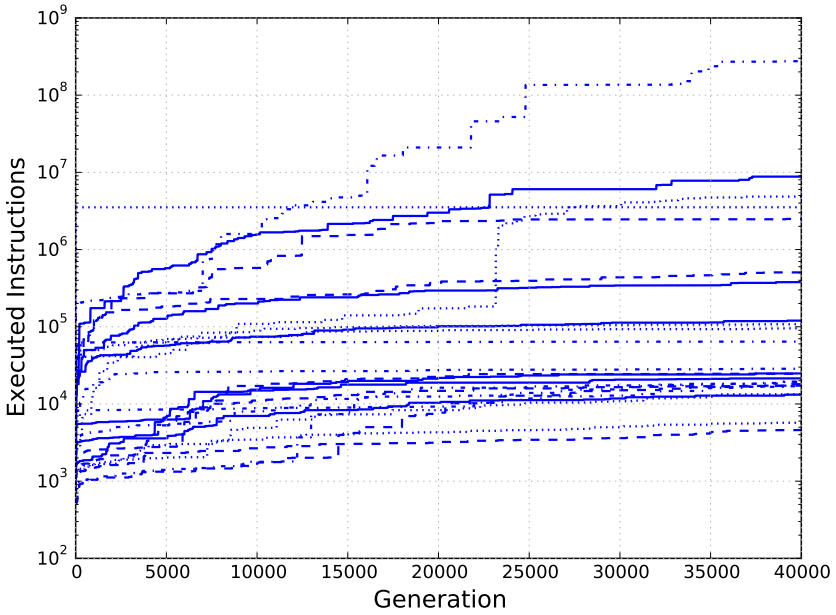

Figure 5.7: Best slowdown achieved by SlowFuzz-generated input strings (normalized over the slowdown of the best-performing input), when matching against fixed regular expressions used in WAFs (normalized against best performing input over an average of 100 runs)

.

Fixed regular expression and mutated string: In the second part of our eval-

[2]Notice that due to SlowFuzz's guidance engine, any regex produced must exhibit increased instruction count as compared to *all* previous regexes.

uation of SlowFuzz on regular expressions, we seek to examine if, for a given *fixed* regular expression, SlowFuzz is able to generate inputs that can introduce a slowdown during matching. We collect PCRE-compliant regular expressions from popular Web Application Firewalls (WAF) [8], and utilized the PCRE library to match input strings generated by SlowFuzz against each regular expression. For this experiment, we apply SlowFuzz on a total of 25 regular expressions, and we record the total instructions executed by the PCRE library when matching the regular expression against SlowFuzz's generated units, at each generation. For our set of regular expressions, SlowFuzz achieved monotonically increasing slowdowns, ranging from 8% to 25%. Figure 5.7 presents how the slowdown varies as fuzzing progresses, for our regex samples.

5.3.3.4 Hash Tables

Hash tables are a core data structure in a wide variety of software. The performance of hash table lookup and insertion operations significantly affects the overall application performance. Complexity attacks against hash table implementations may induce unwanted effects ranging from performance slowdowns to full-fledged DoS [138, 196, 147, 129, 48]. In order to evaluate if SlowFuzz can generate inputs that trigger collisions without knowing any details about the underlying hash functions, we apply it on the hash table implementation of PHP (v5.6.26), which is known to be vulnerable to collision attacks.

PHP Hashtables: Hashtables are prevalent in PHP and they also serve as the backbone for PHP's array interface. PHP v5.x utilizes the DJBX33A hash function for hashing using string keys, which can bee seen in Listing 5.1.

We notice that for two strings of the form 'ab' and 'cd' to collide, the following property must hold [71]:

$$c = a + n \wedge d = b - 33 * n, n \in \mathbb{Z}$$

It is also easy to show that if two equal-length strings A and B collide, then strings xAy, xBy where x and y are any prefix and suffix respectively, also collide. Using the above property, one can construct a worst-case performing sequence of inputs [19], forcing a worst-case insertion time of $O(n^2)$.

```
1 /*
2  * @arKey is the array key to be hashed
3  * @nKeyLenth is the length of arKey
4  */
5 static inline ulong
6 zend_inline_hash_func(const char *arKey, uint nKeyLength)
7 {
8     register ulong hash = 5381;
9
10    for (uint i = 0; i < nKeyLength; ++i) {
11        hash = ((hash << 5) + hash) + arKey[i];
12    }
13
14    return hash;
15 }
```

Listing 5.1: DJBX33A hash without loop unrolling.

Abusing the complexity characteristics of the BJBX33A hash, attackers performed DoS attacks against PHP, Python and Ruby applications in 2011. As a response, PHP added an option in its `ini` configuration to set a limit on the number of collisions that are allowed to happen. However, in 2015, similar DoS attacks [2] were reported, abusing PHP's JSON parsing into hash tables. In this experiment we examine how SlowFuzz performs when applied to this particular hash function implementation.

Our experimental setup is as follows: we ported the PHP hash table implementation so that the latter can be used in any C/C++ implementation, removing all the interpreter-specific variables and macros, however leaving all the non interpreter-related components intact. Subsequently, we created a hash table with a size of 64 entries, and utilized SlowFuzz to perform *a maximum* of 64 insertions to the hash

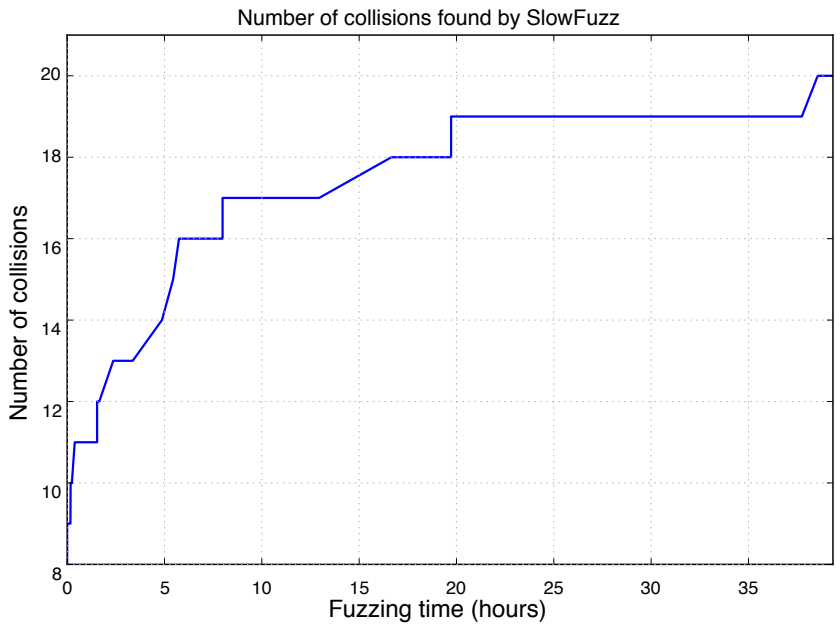

Figure 5.8: Number of collisions found by SlowFuzz per generation, when applying it on the PHP 5.6 hashtable impelemntation, for at most of 64 insertions with string keys.

table, using strings as keys, starting from a corpus consisting of a single input that causes 8 collisions. In particular, the keys for the hash table insertions were provided by SlowFuzz at each generation and SlowFuzz evolved its corpus of strings using a hybrid mutation strategy. Given a hash table of 64 entries and 64 insertions to the hash table, the maximum number of collisions that can be performed is also 64. In order to measure the number of collisions occurring in the hashtable at each generation, we created a PHP module (running in the context of PHP), and measured the number of collisions induced by each input that SlowFuzz generates. We perform our measurements after the respective elements are inserted into a *real* PHP array. Our results are presented in Figure 5.8.

We notice that despite the complex conditions required to trigger a hash collision and without knowing any details about the hash function, SlowFuzz's evolutionary engine reaches 31.25% of the theoretical worst-case after approximately 40 hours of fuzzing, using a single CPU. SlowFuzz's stateful, evolutionary guidance achieves

monotonically increasing slowdowns, despite the complex constraints imposed by the hash function. On the contrary, repeating the same experiment using coverage-based fuzzing, yielded non-monotonically increasing collisions, and at no point an input was generated causing more than 8 collisions. In particular, fuzzing using coverage generated 58 inputs with a median of 5 collisions.

5.3.3.5 ZIP Utilities

Zip utilities that support various compression/decompression schemes are another instance of applications that have been shown to suffer from Denial of Service attacks. For instance, an algorithmic complexity vulnerability used in the sorting algorithms in the `bzip2` application[3] allowed remote attackers to cause DoS via increased CPU consumption, when they provided a file with many repeating inputs [128].

In order to evaluate how SlowFuzz performs when applied to the compression/decompression libraries, we apply it on `bzip2` v1.0.6. In particular, we utilize SlowFuzz to create compressed files of a maximum of 250 bytes, and we subsequently use the libbzip2 library to decompress them. Based on the slowdowns observed during decompression, SlowFuzz evolves its input corpus, mutating each input using its hybrid mode of operation. Our experimental results are presented in Figure 5.9.

A detailed case study: Figure 5.9 depicts the time required by the `bzip2` binary to decompress each of the inputs generated by SlowFuzz. We notice that for the first hour of fuzzing, the inputs generated by SlowFuzz do not exhibit significant slowdown during their decompression by `bzip2`. In particular, each of the 250-byte inputs of SlowFuzz's corpus for the first hour of fuzzing is decompressed in approximately 0.0006 seconds. However, in upcoming generations, we observe that SlowFuzz successfully achieves decompression times reaching 0.18s to 0.21s and an

[3]The vulnerability is found in `BZip2CompressorOutputStream` for Apache Commons Compress before 1.4.1.

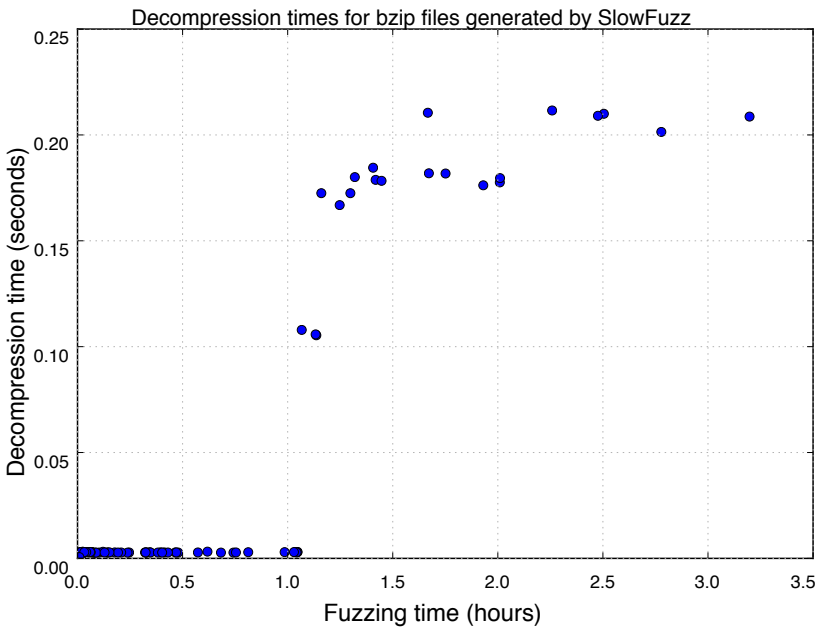

Figure 5.9: Slowdowns observed while decompressing inputs generated by SlowFuzz using the `bzip2` binary. The maximum file size is set to 250 bytes.

overall slowdown in the range of 300x. Particularly, in the first 6 minutes after the first hour, SlowFuzz achieves a decompression time of 0.10 sec. This first peak in the decompression time is achieved because of SlowFuzz triggering the randomization mechanism of `bzip2`, by setting the respective header byte to a non-zero value. This mechanism, although deprecated, was put in place to protect against repetitive blocks, and is still supported for legacy reasons. However, even greater slowdowns are achieved when SlowFuzz mutates two bytes used in `bzip2`'s Move to Front Transform (MTF) [27] and particularly in the run length encoding of the MTF result. Specifically, the mutation of these bytes affects the total number of invocations of the `BZ2_bzDecompress` routine, which results in a total slowdown of 38.31x in decompression time.

The respective code snippet in which the affected bytes are read is shown in Listing 5.2: the `GET_MTF_VAL` macro reads the modified bytes in memory[4]. These

[4]Via the macros `GET_BITS(BZ_X_MTF_3, zvec, zn)` and `GET_BIT(BZ_X_MTF_4, zj)`.

bytes subsequently cause the routine `BZ2_bzDecompress` to be called 4845 times, contrary to a single call before the mutation. We should note at this point, that the total size of the input before and after the mutation remained unchanged.

Finally, in order to compare with a non complexity-targeting strategy, we repeated the previous experiment using traditional coverage-based fuzzing. The fuzzer, when guided only based on coverage, did not generate any input causing executions larger than 0.0008 seconds, with the maximum slowdown achieved being 23.7%.

```
1  do {
2    /* Check that N doesn't get too big, so that
3     es doesn't go negative.  The maximum value
4     that can be RUNA/RUNB encoded is equal
5     to the block size (post the initial RLE),
6     viz, 900k, so bounding N at 2 million
7     should guard against overflow without
8     rejecting any legitimate inputs. */
9    if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
10   if (nextSym == BZ_RUNA) es = es + (0+1) * N; else
11   if (nextSym == BZ_RUNB) es = es + (1+1) * N;
12   N = N * 2;
13   GET_MTF_VAL(BZ_X_MTF_3, BZ_X_MTF_4, nextSym);
14 }
15   while (nextSym == BZ_RUNA || nextSym == BZ_RUNB);
```

Listing 5.2: Excerpt from bzip2's BZ2_decompress routine (decompress.c). A two byte modification by SlowFuzz results in a 38.31x slowdown compared to the previous input.

From the above experiment we observe that SlowFuzz's guidance and mutations engines are successful in pinpointing locations that trigger large slowdowns even in very complex applications such as a state-of-the-art compression utility like `bzip2`.

5.3.3.6 Engine Evaluation

Effect of SlowFuzz's fitness function: In this section, we examine the effect of using code-coverage-guided search versus SlowFuzz's resource usage based fitness function, particularly in the context of scanning an application for complexity vulnerabilities. To do so, we repeat one of the experiments of Section 5.3.3.2, applying SlowFuzz on the OpenBSD quicksort implementation with an input size of 64 bytes, for a total of 1 million generations, using hybrid mutations. Our results are presented in Figure 5.10. We observe that SlowFuzz's guidance mechanism yields significant improvement over code-coverage-guided search. In particular, SlowFuzz achieves a 3.3x slowdown for OpenBSD, whereas the respective slowdown achieved using only coverage-guided search is 23.41%. This is an expected result, since, as mentioned in previous Sections, code coverage cannot encapsulate behaviors resulting in multiple invocations of the same line of code (e.g., an infinite loop). Moreover, we notice that the total instructions of each unit that is created by SlowFuzz at different generations is not monotonically increasing. This is an artifact of our implementation, using SanitizerCoverage's 8-bit counters, which provide a coarse-grained, imprecise tracking of the real number of times each edge was invoked (Section 5.3.2). Thus, although a unit might result in execution of fewer instructions, it will only be observed by SlowFuzz's guidance engine if the respective number of total CFG edges falls into a separate bucket (8 possible ranges representing the total number of CFG edge accesses). Future work can consider applying more precise instruction tracking (e.g., using hardware counters or utilities similar to `perf`) with static analyses passes, to achieve more effective guidance.

Finally, when choosing SlowFuzz's fitness function, we also considered the option of utilizing time-based tracking instead of performance counters. However, performing time-based measurements in real-world systems is not trivial, especially at instruction-level granularity and when multiple samples are required in order to min-

imize measurement errors. In the context of fuzzing, multiple runs of the same input will slow the fuzzer down significantly. To demonstrate this point, in Figure 5.10, we also include an experiment in which the execution time of an input is used to guide input generation. In particular, we utilized CPU clock time to measure the execution time of a unit and discarded the unit if it was not slower than all previously seen units. We notice that the corpus degrades due to system noise and does not achieve any slowdown larger than 23%.[5]

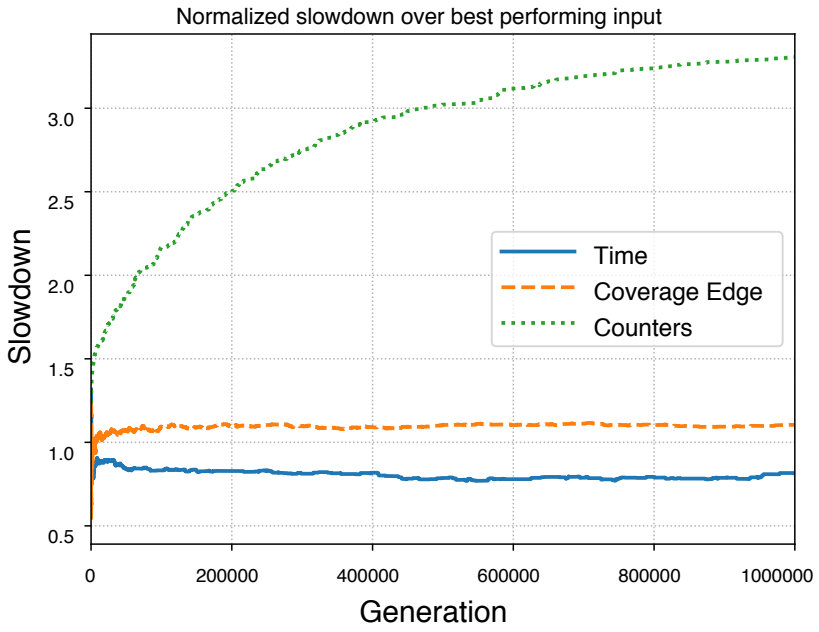

Figure 5.10: Comparison of the slowdown achieved by SlowFuzz under different guidance mechanisms, when applied on the OpenBSD quicksort implementation of Section 5.3.3.2, for an input size of 64 bytes, after 1 million generations (average of 100 runs).

Effect of mutation schemes: To highlight the different characteristics of each of SlowFuzz's mutation schemes described in Section 5.3.1, we repeat one of the experiments of Section 5.3.3.2, applying SlowFuzz on the OpenBSD quicksort, each time using a different mutation strategy. Our experimental setup is identical with

[5]Contrary to the slowdowns measured during fuzzing using a single run, the slowdowns presented in Figure 5.10 are generated using the `perf` utility running ten iterations per input. Non-monotonic increases denote corpus degradation due to bad input selection.

that of Section 5.3.3.2: we sort inputs with a size of 64 bytes and fuzz for a total of 1 million generations. For each mode of operation, we average on a total of 100 SlowFuzz sessions. Our results are presented in Figure 5.11.

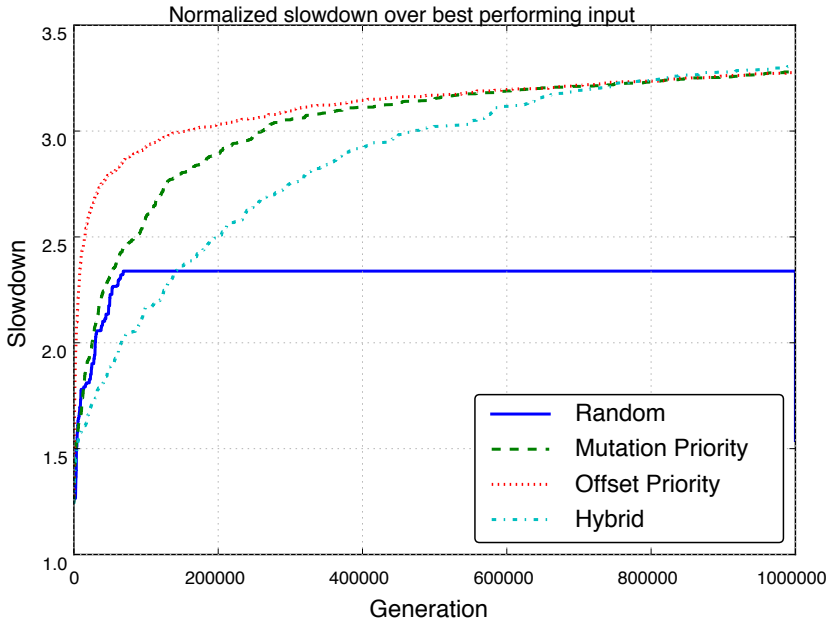

Figure 5.11: Comparison of the best slowdown achieved by SlowFuzz's different mutation schemes, at each generation, when applied on the OpenBSD quicksort implementation of Section 5.3.3.2, for an input size of 64 bytes, after 1 million generations (average of 100 runs).

We notice that, for the above experiment, choosing a mutation at random, is the worst performing option among all mutation options supported by SlowFuzz (Section 5.3.1.2), however still achieving a slowdown of 2.33x over the best performing input. Indeed, all of SlowFuzz's scoring-based mutation engines (offset-priority, mutation-priority and hybrid), are expected to perform at least as good as selecting mutations at random, given enough generations, as they avoid getting stuck with unproductive mutations. We also observe that offset priority is the fastest mode to converge out of the other mutation schemes for this particular experiment, and results in an overall slowdown of 3.27x.

For sorting, offsets that correspond to areas of the array that should *not* be mu-

tated, are quickly penalized under the offset priority scheme, thus mutations are mainly performed on the non-sorted portions of the array. Additionally, we observe that mutation priority also outperforms the random scheme due to the fact that certain mutations (e.g., crossover operations) may have devastating effects on the sorting of the array. The mutation priority scheme picks up such patterns and avoids such mutations. By contrast, these mutations continue to be used under the random scheme. Finally, we observe that the hybrid mode eventually outperforms all other strategies, achieving a 3.30x slowdown, however is the last mutation mode to start reaching a plateau. We suspect that this results from the fact the hybrid mode does not quickly penalize particular inputs or mutations as it needs more samples of each mutation operation and offset pair before avoiding any particular offset or mutation operation.

Instrumentation overhead: SlowFuzz's runtime overhead, measured in executions per second, matches the overhead of native libFuzzer. The executions per second achieved on different payloads are mostly dominated by the runtimes of the native binary, as well as the respective I/O operations. Despite our choice to prototype SlowFuzz using libFuzzer, the design and methodology presented in Section 5.3.1 can be applied to any evolutionary fuzzer and can also be implemented using Dynamic Binary Instrumentation frameworks, such as Intel's PIN [108], to allow for more detailed runtime tracking of the application state. However, such frameworks are known to incur slowdowns of more than 200%, even with minimal instrumentation [133]. For instance, for our PHP hashtable experiments described in Section 5.3.3.4, an insertion of 16 strings, resulting in 8 collisions, takes 0.02 seconds. Running the same insertion under a PIN tool that only counts instructions, requires a total of ~2 seconds. By contrast, hashtable fuzzing with SlowFuzz achieves up to 4000 execs/sec, unless a significant slowdown is incurred due to a particular input.[6]

[6]Execution under SlowFuzz does not require repeated loading of the required libraries, but is

5.4 Discussion

In this Chapter, we demonstrated that evolutionary search techniques commonly used in fuzzing to find memory safety bugs can be adapted to find algorithmic complexity vulnerabilities. Similar strategies should be applicable for finding other types of DOS attacks like battery draining, filling up memory or hard disk, etc. Designing the fitness functions and mutation schemes for detecting such bugs will be an interesting future research problem. Besides evolutionary techniques, using other mechanisms like Reinforcement Learning or Monte Carlo search can also be adapted for finding inputs with worst-case resource usage.

The current prototype implementation of SlowFuzz uses the SanitizerCoverage passes to keep track of the number of times a CFG edge is accessed. Such tracking is limited by the total number of buckets allowed by SanitizerCoverage. This reduces the accuracy of resource usage information as tracked by SlowFuzz since any edge that is accessed more than 128 times is assigned to the same bucket regardless of the actual number of accesses. Although, under its current implementation, the actual edge count information is imprecise, this is not a fundamental design limitation of SlowFuzz but an artifact of our prototype implementation. Alternative implementations can offer more precise tracking via custom callbacks for SanitizerCoverage, by adopting hardware counters or by utilizing per-unit `perf` tracking.[7] Moreover, similar techniques can be used to target vulnerabilities regarding space or memory complexity. However, in this case, it is necessary to perform fine-grained tracking of operations such as heap allocations, file creations, etc. which we did not explore in this work. Compiler-inserted monitors can be combined, towards this end, with dynamic modules (e.g., runtime libraries), that can pinpoint the patterns in space

only dominated by the function being tested, which is only a fraction of the total execution of the native binary (thus smaller than 0.02 seconds).

[7]The benefit, on the other hand, of the current implementation is that it can be incorporated into libFuzzer's main engine orthogonally, without requiring major changes to libFuzzer's dependencies.

usage of the tested binary, and adjust input generation accordingly.

Although lines of work related with Worst-Case Execution Time (WCET) estimation [180, 137, 136, 17, 16, 195] fall outside our analysis scope, the techniques presented in this Chapter can orthogonally be combined with other frameworks targeting performance bugs. For instance, recent work by Holland et al. [73] combines static and dynamic analysis to perform analyst-driven exploration of Java programs to detect complexity vulnerabilities. Contrary to SlowFuzz, this work requires a human analyst to closely guide the exploration process, specifying which portions of the binary should be analyzed statically and which dynamically as well as defining the inputs to the binary. Integrating SlowFuzz into such a system would allow for efficient human-assisted fuzzing, where an analyst can specify the contexts and application components in which SlowFuzz would be most effective.

Integrating existing static analysis or hybrid techniques into SlowFuzz can further improve its performance. Using static analysis to find potentially promising offsets in an input for mutation will further reduce the search space and therefore will make the search process more efficient. Along these lines, Lu et al. study a large set of real-world performance bugs to construct a set of rules that they use to discover new performance bugs via hand-built checkers integrated in the LLVM compiler infrastructure [86]. LDoctor [164] detects loop inefficiencies by implementing a hybrid static-dynamic program analysis that leverages different loop-specific rules. Both the above lines of work, contrary to SlowFuzz, require expert-level knowledge for creating the detection rules, and can be orthogonally be combined with it. Another line of work that could be combined with the techniques presented in this Chapter focuses on application profiling to detect performance bottlenecks: Ramanathanet al. utilize flow profiling for the efficient detection of memory-related performance bugs in Java programs [120], whereas Grechanik et al. utilize a genetic-algorithm-driven profiler for detecting performance bottlenecks [156] in Web applications, and cluster

execution traces to explore different combinations of the input parameter values.

Chapter 6

Conclusion

6.1 Summary

In this dissertation we investigated the hypothesis that context-aware, adaptive analyses can increase the accuracy, re-usability and scope of existing binary application testing toolchains, as well as enable us to set the foundations for generally applicable binary testing frameworks.

Towards this goal, we initially examined the first point of our hypothesis, i.e., how context-aware analyses can allow for more accurate detection and reporting of errors, and we presented a methodology under which current compilers can be extended to prioritize errors based on their criticality, achieving more targeted static analyses. We applied this methodology in the LLVM compiler toolchain, focusing on integer errors as a use-case: applying information flow tracking, we augmented the compiler pipeline to insert dynamic monitors to the compiled binary so as to report integer errors by taking into account the context in which the errors appear in. Using the proposed technique, the compiler no longer reports errors indiscriminately (agnostically to the context they appear in), but instead appropriately prioritizes them based on their likelihood to have undesired implications in the program execution.

Subsequently, we examined how context-aware analyses that support state prioritization can provide adaptive testing, targeting broader classes of errors and enabling the construction of modular infrastructure components that can be re-used across different toolchains. To this end, we make several contributions, advancing the state-of-the-art in compiler-assisted feedback-driven testing: Initially we introduce differential diversity (δ-diversity) and present a methodology for achieving context-aware guidance in feedback-driven toolchains. In the following, we demonstrate how context-aware techniques can augment the different components (i.e., instrumentation, analysis & mutation engines) of modern evolutionary fuzzing frameworks, so that the latter can target different types of bugs selectively. Particularly, we present the first, to the best of our knowledge, prototypes of generic evolutionary fuzzers targeting, additionally to crash-inducing bugs, logic errors and complexity vulnerabilities. We demonstrate that utilizing the proposed adaptive design paradigms, the same fuzzer be can retrofitted to successfully target different types of errors, in different real-world applications, with completely different characteristics, depending on the context of the analysis. Although, for our prototype implementations, we extend a production-level fuzzer to target two new classes of errors, the techniques presented are not limited to the presented error classes (i.e., complexity and logic bugs), but instead can be applied to different feedback-driven toolchains, as well as different classes of errors.

6.2 Discussion

We hope that this work presents a solid argument for binary application testing frameworks to abandon traditional monolithic design primitives, and instead adopt context-aware analyses that allow for agile, adaptive testing. Since context-aware designs can be orthogonally applied to existing infrastructure, they can gradually be

utilized in legacy frameworks, increasing the modularity and interoperability of the respective toolchains. Augmenting the current software ecosystem to support adaptive analyses necessitates that the proper design abstractions are in place, separating the context-specific and context-agnostic components of each testing framework. As such, larger adoption of these designs will enable re-usability of context-agnostic modules, as well as broaden the use of context-specific modules to different applications that may face similar constraints. For instance, the same problem of prioritization that fuzzers face with respect to input generation, is encountered in symbolic execution engines with respect to constraint solving prioritization. Modules and algorithms targeting state prioritization according to particular attributes can be developed, if the appropriate abstractions are in place, for the general case, in a context-agnostic manner, and shared across different testing frameworks. Such a task is no small feat; however we hope that the techniques proposed in this work can lay the foundations for more research in this direction. In the following Section we outline some fruitful future venues for research, extending this line of work.

6.3 Future Directions

6.3.1 Semantic Abstractions

Designing and implementing generally applicable testing frameworks is a hard problem. An obstacle prohibiting the development of such generic toolchains is the lack of appropriate abstractions, allowing to encapsulate the semantics of the applications being tested. Despite the fact that this problem is, in general, intractable [39, 153, 26, 207, 51], there are several subproblems that can be attempted, using existing knowledge. Unfortunately little progress has been made in the direction of modelling the semantic properties of software and integrating these properties into the testing ecosystem in an automated manner. As such, it is currently infeasible to construct,

in the generic case, toolchains that allow for transfer of knowledge, not solely across testing sessions, but, more importantly, across testing of different applications in different deployments. For instance, as discussed in our Introduction, a fuzzer that is applied to a particular PDF viewer, does not "learn" properties that might be useful in fuzzing, say, different PDF viewers, or completely different applications such as a text editor. On the contrary, experienced human analysts manage to recognize common erroneous patterns when auditing previously unseen software, based on their past experience.

Fortunately, the current status quo allows for research advances in this direction, that is, towards such automated memoization of testing invariants/properties. For example, one can apply state-of-the-art testing tools to open-source software, with known vulnerabilities, and attempt to "learn" (e.g., utilizing neural networks), what testing abstractions are successful in locating certain bug patterns, in an application-agnostic manner, so as to apply this knowledge in new, previously unseen, applications. To make this example clearer, let us consider the popular case of combining fuzzing and symbolic execution for testing, and how we can attempt to generalize a framework combining the two techniques. A key observation is that, both fuzzers and symbolic executors are completely agnostic to the format of real workloads, and cannot make informed decisions during testing based on past executions or different programs. Our goal thus, is to train our testing frameworks' engines so that they behave similarly to a human analyst. To do so, we need an understanding of the following: i) what are the characteristics of different bug types from the perspective of the automated framework (i.e., an integer overflow will always affect/depend upon particular types of instructions – can we detect this examining only the binary? what about if we had support from the compiler or a hypervisor?) ii) what are the expected inputs/state transitions for a particular application under normal workloads and what components are shared across many applications (i.e., can we learn what a

server application architecture looks like? can we distinguish what parts of an application's code are simple copies from an online repository?) iii) which, out of all the available known testing methodologies, will be the ones that are predicted to be most effective for the given application being tested (based on our models from points (i) & (ii))?.

The quantity of today's open-source software [63, 18, 68] allows for vast code aggregation and clustering at scale. For instance, one may study what are the properties of common bugs by examining past CVEs and by profiling the respective applications with real workloads, training a model which will be, with the appropriate abstractions, application-agnostic, and will encapsulate the properties of given bug types. Such training can allow for reasoning about what are the most likely successful steps the testing framework can take in order to trigger different bugs. There exists related work addressing some of the points mentioned above, in isolation. Future work may focus on developing abstractions connecting the information of the aforementioned sub-categories into a unified, automated, framework.

6.3.2 "Old" is the New "New"

Computer Science is a fairly new field, compared to, say, Mathematics or Physics. However, in its brief history, it has seen tremendous advances and nowadays impacts every aspect of human life. An interesting phenomenon throughout the few decades that form the history of computing[1] is that a plethora of techniques are constantly being "reinvented", with minor or major alterations, and re-applied in "new" settings. Such reinventions are often very impactful, since the status quo from the era under which a technique was formed may have changed so drastically, that the availability of additional toolchains, analyses and resources may dramatically improve the results of the performed analyses.

[1] A phenomenon not limited to this particular field.

As such, drawing from the current line of work, it will be fruitful to revisit techniques proposed in the past and augment them accordingly with novel concepts. For instance, one may apply the differential testing methodologies presented in Chapter 4 to techniques presented in different domains such as this of system administration and fault detection and recovery [13, 142].

More importantly, however, we should reflect on the history of software, to plan for its future: one may consider, for instance, the implications of deploying *en masse*, in the first decades of the software boom, of unsafe code, and the disproportionate development of testing infrastructure and how that impacted the current status quo. This knowledge can help us, as a community, to plan ahead, when deploying at scale new technologies, such as neural networks or smart contracts, so that we don't rush to develop testing solutions post-mortem, but rather provide guarantees by design, at the very creation of each new technology.

Bibliography

[1] *'$300m in cryptocurrency' accidentally lost forever due to bug | Technology | The Guardian*. https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether. (Accessed on 01-18-2018).

[2] *#800564 - PHP5: trivial hash complexity DoS attack - Debian Bug report logs*. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=800564.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10088-6.

[4] Periklis Akritidis et al. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." In: *USENIX Security Symposium*. 2009, pp. 51–66.

[5] Periklis Akritidis et al. "Preventing memory error exploits with WIT." In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 263–277.

[6] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. "BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs)." In: *Digital Investigation* 18 (2016), S11–S22.

[7] George Argyros et al. "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2016, pp. 1690–1701.

[8] *attackercan/regexp-security-cheatsheet*. https://github.com/attackercan/regexp-security-cheatsheet/tree/master/RegexpSecurityParser/WAF-regexps.

[9] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. "A survey on context-aware systems." In: *International Journal of Ad Hoc and Ubiquitous Computing* 2.4 (2007), pp. 263–277.

[10] Osbert Bastani et al. "Synthesizing program input grammars." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2017, pp. 95–110.

[11] Gabriele Bavota et al. "When does a refactoring induce bugs? an empirical study." In: *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE. 2012, pp. 104–113.

[12] Ilan Beer et al. "RuleBase: An Industry-oriented Formal Verification Tool." In: *Proceedings of the 33rd Annual Design Automation Conference*. DAC '96. Las Vegas, Nevada, USA: ACM, 1996, pp. 655–660. ISBN: 0-89791-779-0. DOI: 10.1145/240518.240642. URL: http://doi.acm.org/10.1145/240518.240642.

[13] Steven M. Bellovin and Randy Bush. "Configuration Management and Security." In: *IEEE Journal on Selected Areas in Communications* 27.3 (2009), pp. 268–274. URL: https://www.cs.columbia.edu/~smb/papers/config-jsac.pdf.

[14] Azzedine Benameur, Nathan S. Evans, and Matthew C. Elder. "MINESTRONE: Testing the SOUP." In: *Proceedings of CSET*. Washington, D.C.: USENIX, 2013. URL: https://www.usenix.org/conference/cset13/workshop-program/presentation/Benameur.

[15] Martin Berglund, Frank Drewes, and Brink van der Merwe. "Analyzing catastrophic backtracking behavior in practical regular expression matching." In: *arXiv preprint arXiv:1405.5599* (2014).

[16] Guillem Bernat, Antoine Colin, and Stefan M Petters. "WCET analysis of probabilistic hard real-time systems." In: *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*. IEEE. 2002, pp. 279–288.

[17] Adam Betts, Nicholas Merriam, and Guillem Bernat. "Hybrid measurement-based WCET analysis at the source level using object-level traces." In: *OASIcs-OpenAccess Series in Informatics*. Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[18] *Bitbucket | The Git solution for professional teams*. https://bitbucket.org/. (Accessed on 02-22-2018).

[19] *bk2204/php-hash-dos: A PoC hash complexity DoS against PHP*. https://github.com/bk2204/php-hash-dos.

[20] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain." In: *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria, 2016, pp. 1–12.

[21] Robert S Boyer, Bernard Elspas, and Karl N Levitt. "SELECT—a formal system for testing and debugging programs by symbolic execution." In: *ACM SigPlan Notices* 10.6 (1975), pp. 234–245.

[22] Chad Brubaker et al. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations." In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2014, pp. 114–129.

[23] D Brumley et al. "RICH: Automatically Protecting Against Integer-Based Vulnerabilities." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2007.

[24] David Brumley et al. "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation." In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Boston, MA: USENIX Association, 2007.

[25] Gregory Buehrer, Bruce W Weide, and Paolo AG Sivilotti. "Using parse tree validation to prevent SQL injection attacks." In: *Proceedings of the 5th international workshop on Software engineering and middleware*. ACM. 2005, pp. 106–113.

[26] Leslie Burkholder. "The halting problem." In: *ACM SIGACT News* 18.3 (1987), pp. 48–60.

[27] *bzip2*. http://www.bzip.org/1.0.3/html/index.html.

[28] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[29] Cristian Cadar and Dawson Engler. "Execution generated test cases: How to make systems code crash itself." In: *International SPIN Workshop on Model Checking of Software*. Springer. 2005, pp. 2–23.

[30] Xiang Cai, Yuwei Gui, and Rob Johnson. "Exploiting Unix file-system races via algorithmic complexity attacks." In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE. 2009, pp. 27–41.

[31] Cristiano Calcagno et al. "Moving fast with software verification." In: *NASA Formal Methods Symposium*. Springer. 2015, pp. 3–11.

[32] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 147–160.

[33] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. "An industrial survey on contemporary aspects of software testing." In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE. 2010, pp. 393–401.

[34] S. K. Cha, M. Woo, and D. Brumley. "Program-Adaptive Mutational Fuzzing." In: *2015 IEEE Symposium on Security and Privacy (S&P)*. 2015, pp. 725–741.

[35] Peter Chapman and David Evans. "Automated black-box detection of side-channel vulnerabilities in web applications." In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 263–274.

[36] Liming Chen and Algirdas Avizienis. "N-version programming: A fault-tolerance approach to reliability of software operation." In: *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*. 1978, pp. 3–9.

[37] Yuting Chen and Zhendong Su. "Guided differential testing of certificate validation in SSL/TLS implementations." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 793–804.

[38] Yuting Chen et al. "Coverage-directed differential testing of JVM implementations." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. 2016, pp. 85–99.

[39] Brian Chess and Gary McGraw. "Static analysis for security." In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79.

[40] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: A compile-time solution to buffer overflow attacks." In: *Distributed Computing Systems, 2001. 21st International Conference on*. IEEE. 2001, pp. 409–417.

[41] Zheng Leong Chua et al. "Neural Nets Can Learn Function Type Signatures From Binaries." In: *Proceedings of the 26th USENIX Conference on Security Symposium, Security*. Vol. 17. 2017.

[42] Thomas H.. Cormen et al. *Introduction to algorithms*. Vol. 6. MIT press Cambridge, 2001.

[43] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252.

[44] Crispin Cowan et al. "Protecting systems from stack smashing attacks with StackGuard." In: *Linux Expo*. 1999.

[45] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks." In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 3–3.

[46] *CVE-2011-5021*. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2011-5021.

[47] *CVE-2013-2099*. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2013-2099.

[48] *CVE-2015-2526*. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2015-2526.

[49] *CWE - Common Weakness Enumeration*. http://cwe.mitre.org/.

[50] Ermira Daka and Gordon Fraser. "A survey on unit testing practices and problems." In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE. 2014, pp. 201–211.

[51] Alexander W Dent. "Fundamental problems in provable security and cryptography." In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 364.1849 (2006), pp. 3215–3230.

[52] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. "A Comprehensive Study of Real-world Numerical Bug Characteristics." In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 509–519. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155627.

[53] Will Dietz et al. "Understanding integer overflow in C/C++." In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 2012.

[54] *Dillo Web Browser :: Home Page*. https://www.dillo.org/. (Accessed on 02-08-2018).

[55] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[56] Manuel Egele et al. "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components." In: *Usenix Security*. 2014, pp. 303–317.

[57] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. "discovre: Efficient cross-architecture identification of bugs in binary code." In: *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*. 2016.

[58] Asger Feldthaus and Anders Møller. "Checking correctness of TypeScript interfaces for JavaScript libraries." In: *ACM SIGPLAN Notices*. Vol. 49. 10. ACM. 2014, pp. 1–16.

[59] Qian Feng et al. "Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis." In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM. 2016, pp. 11–22.

[60] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software." In: *ACM Sigplan Notices*. Vol. 40. 1. ACM. 2005, pp. 110–121.

[61] Vahid Garousi et al. "What industry wants from academia in software testing?: Hearing practitioners' opinions." In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM. 2017, pp. 65–69.

[62] *GIMP - GNU Image Manipulation Program*. https://www.gimp.org/. (Accessed on 02-08-2018).

[63] *GitHub*. https://github.com/. (Accessed on 02-22-2018).

[64] *gnulib/qsort.c at master coreutils/gnulib*. https://github.com/coreutils/gnulib/blob/master/lib/qsort.c.

[65] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-based Whitebox Fuzzing." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Tucson, AZ, USA, 2008, pp. 206–215.

[66] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[67] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *Proceedings of the 2008 Network and Distributed Systems Symposium (NDSS)*. Vol. 8. 2008, pp. 151–166.

[68] *Google Code*. https://code.google.com/. (Accessed on 02-22-2018).

[69] Robert Guo. "MongoDB's JavaScript Fuzzer." In: *Commun. ACM* 60.5 (Apr. 2017), pp. 43–47. ISSN: 0001-0782. DOI: 10.1145/3052937. URL: http://doi.acm.org/10.1145/3052937.

[70] Istvan Haller et al. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations." In: *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 49–64.

[71] *Hash algorithm and collisions - PHP Internals Book*. http://www.phpinternalsbook.com/hashtables/hash_algorithm.html.

[72] Chris Hawblitzel et al. "IronFleet: proving practical distributed systems correct." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, pp. 1–17.

[73] Benjamin Holland et al. "Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities." In: *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE. 2016, pp. 79–84.

[74] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with code fragments." In: *21st USENIX Security Symposium (USENIX Security '12)*. 2012, pp. 445–458.

[75] Allen D Householder and Jonathan M Foote. "Probability-based parameter selection for black-box fuzz testing." In: *CMU/SEI Technical Report - CMU/SEI-2012-TN-019*. 2012.

[76] *https://opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/kern/qsort.c*. https://opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/kern/qsort.c.

[77] Koray Incki, Ismail Ari, and Hasan Sözer. "A survey of software testing in the cloud." In: *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on*. IEEE. 2012, pp. 18–23.

[78] *Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile*. http://tools.ietf.org/html/rfc5280. 2008.

[79] *IOActive_ELF_Parsing_with_Melkor.pdf*. http://www.ioactive.com/pdfs/IOActive_ELF_Parsing_with_Melkor.pdf.

[80] *IRS computer glitch costs U.S. millions - Houston Chronicle*. http://www.chron.com/news/nation-world/article/IRS-computer-glitch-costs-U-S-millions-1660750.php. (Accessed on 01-18-2018).

[81] *Isartor Test Suite (Terms of Use & Download) - PDF Association*. https://www.pdfa.org/isartor-test-suite-terms-of-use-download/.

[82] Timur Iskhodzhanov et al. "ThreadSanitizer, MemorySanitizer." In: ().

[83] Suman Jana and Vitaly Shmatikov. "Abusing File Processing in Malware Detectors for Fun and Profit." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2012, pp. 80–94.

[84] Suman Jana et al. "Automatically Detecting Error Handling Bugs using Error Specifications." In: *25th USENIX Security Symposium (USENIX Security)*. Austin, 2016.

[85] Kangkook Jee. *On Efficiency and Accuracy of Data Flow Tracking Systems*. Columbia University, 2016.

[86] Guoliang Jin et al. "Understanding and detecting real-world performance bugs." In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 77–88.

[87] Richard WM Jones and Paul HJ Kelly. "Backwards-compatible bounds checking for arrays and pointers in C programs." In: *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. 001. Linköping University Electronic Press. 1997, pp. 13–26.

[88] Yuan Kang, Baishakhi Ray, and Suman Jana. "APEx: Automated Inference of Error Specifications for C APIs." In: *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Singapore, 2016.

[89] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-User Attacks." In: *USENIX Security Symposium*. Vol. 16. 2012.

[90] James C King. "Symbolic execution and program testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[91] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. "Static analysis for regular expression denial-of-service attacks." In: *International Conference on Network and System Security*. Springer. 2013, pp. 135–148.

[92] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel." In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[93] *Knight Shows How to Lose $440 Million in 30 Minutes - Bloomberg*. https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes. (Accessed on 01-18-2018).

[94] Jesse Kornblum. "Identifying almost identical files using context triggered piecewise hashing." In: *Digital investigation* 3 (2006), pp. 91–97.

[95] Volodymyr Kuznetsov et al. "Code-Pointer Integrity." In: *OSDI*. Vol. 14. 2014, p. 00000.

[96] Pavel Laskov et al. "Practical evasion of a learning-based classifier: A case study." In: *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2014, pp. 197–211.

[97] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the international symposium on Code generation and optimization (CGO)*. 2004.

[98] Chris Lattner, Andrew Lenharth, and Vikram Adve. "Making context-sensitive points-to analysis with heap cloning practical for the real world." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2007.

[99] Vu Le, Chengnian Sun, and Zhendong Su. "Finding deep compiler bugs via guided stochastic program mutation." In: *ACM SIGPLAN Notices*. Vol. 50. 10. ACM. 2015, pp. 386–399.

[100] Juneyoung Lee et al. "Taming Undefined Behavior in LLVM." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 633–647. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062343. URL: http://doi.acm.org/10.1145/3062341.3062343.

[101] Xavier Leroy. "Formal Verification of a Realistic Compiler." In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814.

[102] Nancy Leveson et al. "Medical devices: The therac-25." In: *Appendix of: Safeware: System Safety and Computers* (1995).

[103] Jinku Li et al. "Defeating return-oriented rootkits with return-less kernels." In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 195–208.

[104] *libc/stdlib/qsort.c*. https://sourceforge.net/u/lluct/me722-cm/ci/f3ae3e66860629a7ebe223fdda3fdc8ffbdd9c6d/tree/bionic/libc/stdlib/qsort.c.

[105] *libFuzzer - a library for coverage-guided fuzz testing - LLVM 3.9 documentation*. http://llvm.org/docs/LibFuzzer.html.

[106] Fan Long et al. "Sound input filter generation for integer overflow errors." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2014.

[107] Nuno P. Lopes et al. “Practical Verification of Peephole Optimizations with Alive.” In: *Commun. ACM* 61.2 (Jan. 2018), pp. 84–91. ISSN: 0001-0782. DOI: 10.1145/3166064. URL: http://doi.acm.org/10.1145/3166064.

[108] Chi-Keung Luk et al. “Pin: building customized program analysis tools with dynamic instrumentation.” In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[109] Lannan Luo et al. “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection.” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 389–400.

[110] Brian A Malloy and James F Power. “An interpretation of Purdom’s algorithm for automatic generation of test cases.” In: (2001).

[111] Darko Marinov and Sarfraz Khurshid. “TestEra: A novel framework for automated testing of Java programs.” In: *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE. 2001, pp. 22–31.

[112] Ali Jose Mashtizadeh et al. “CCFI: cryptographically enforced control flow integrity.” In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 941–951.

[113] Nicholas D Matsakis and Felix S Klock II. “The rust language.” In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[114] Peter M. Maurer. “Generating test data with enhanced context-free grammars.” In: *Ieee Software* 7.4 (1990), pp. 50–55.

[115] M. Douglas McIlroy. “A killer adversary for quicksort.” In: *Softw., Pract. Exper.* 29.4 (1999), pp. 341–344.

[116] William M. McKeeman. “Differential Testing for Software.” In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[117] Leo A Meyerovich and Ariel S Rabkin. "Empirical analysis of programming language adoption." In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 1–18.

[118] Barton P Miller, Louis Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities." In: *Communications of the ACM* 33.12 (1990), pp. 32–44.

[119] Scott Moore. *thinkmoore/llvm-deps*. https://github.com/thinkmoore/llvm-deps. (Visited on 06/07/2014).

[120] Rashmi Mudduluru and Murali Krishna Ramanathan. "Efficient flow profiling for detecting performance bugs." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 413–424.

[121] Madanlal Musuvathi et al. "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *OSDI*. Vol. 8. 2008, pp. 267–280.

[122] Santosh Nagarakatte et al. "CETS: compiler enforced temporal safety for C." In: *ACM Sigplan Notices*. Vol. 45. 8. ACM. 2010, pp. 31–40.

[123] Santosh Nagarakatte et al. "SoftBound: Highly compatible and complete spatial memory safety for C." In: *ACM Sigplan Notices* 44.6 (2009), pp. 245–258.

[124] Kedar Namjoshi and Girija Narlikar. "Robust and fast pattern matching for intrusion detection." In: *INFOCOM, 2010 Proceedings IEEE*. IEEE. 2010, pp. 1–9.

[125] *NetBSD: qsort.c,v 1.13 2003/08/07*. http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/stdlib/qsort.c.

[126] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavy-weight dynamic binary instrumentation." In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[127] *Northeast blackout of 2003 - Wikipedia*. https://en.wikipedia.org/wiki/Northeast_blackout_of_2003. (Accessed on 01-18-2018).

[128] *NVD - CVE-2012-2098*. https://nvd.nist.gov/vuln/detail/{CVE}-2012-2098.

[129] *NVD - CVE-2013-4287*. https://nvd.nist.gov/vuln/detail/{CVE}-2013-4287.

[130] Kaan Onarlioglu et al. "G-Free: defeating return-oriented programming through gadget-less binaries." In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM. 2010, pp. 49–58.

[131] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. "Test-data generation using genetic algorithms." In: *Software Testing Verification and Reliability* 9.4 (1999), pp. 263–282.

[132] *PCRE - Perl Compatible Regular Expressions*. http://www.pcre.org/.

[133] Theofilos Petsios et al. "Dynaguard: Armoring canary-based protections against brute-force attacks." In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 351–360.

[134] Theofilos Petsios et al. "NEZHA: Efficient Domain-Independent Differential Testing." In: *Proceedings of the 38th IEEE Symposium on Security & Privacy,(San Jose, CA)*. 2017.

[135] Theofilos Petsios et al. "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 2155–2168.

[136] Stefan M Petters. "Bounding the execution time of real-time tasks on modern processors." In: *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*. IEEE. 2000, pp. 498–502.

[137] Stefan M Petters and Georg Farber. "Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible." In: *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE. 1999, pp. 442–449.

[138] *PHP Vulnerability May Halt Millions of Servers - PHP Classes*. https://www.phpclasses.org/blog/post/171-PHP-Vulnerability-May-Halt-Millions-of-Servers.html.

[139] *Pidgin, the universal chat client*. https://pidgin.im/. (Accessed on 02-08-2018).

[140] Marios Pomonis et al. "IntFlow: improving the accuracy of arithmetic error detection using information flow tracking." In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM. 2014, pp. 416–425.

[141] Marios Pomonis et al. "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 420–436.

[142] Shaya Potter, Steven M. Bellovin, and Jason Nieh. "Two Person Control Administration: Preventing Administration Faults through Duplication." In: *LISA '09*. 2009. URL: http://www.usenix.org/events/lisa09/tech/full_papers/potter.pdf.

[143] David A Ramos and Dawson Engler. "Under-constrained symbolic execution: correctness checking for real code." In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 49–64.

[144] David A Ramos and Dawson R Engler. "Practical, low-effort equivalence verification of real code." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 669–685.

[145] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[146] Baishakhi Ray et al. "A large scale study of programming languages and code quality in github." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 155–165.

[147] *Regular expression Denial of Service - ReDoS - OWASP*. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[148] Jesse Ruderman. *Introducing jsfunfuzz*. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[149] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector." In: *NDSS*. Vol. 2004. 2004, pp. 159–169.

[150] Michele Sama et al. "Context-aware adaptive applications: Fault patterns and their automated identification." In: *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 644–661.

[151] *SantizerCoverage - Clang 4.0 documentation*. http://clang.llvm.org/docs/SanitizerCoverage.html.

[152] *SantizerCoverage - Clang 4.0 documentation*. http://clang.llvm.org/docs/SanitizerCoverage.html.

[153] Len Sassaman et al. "The halting problems of network stack insecurity." In: ().

[154] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX Annual Technical Conference*. 2012, pp. 309–318.

[155] Konstantin Serebryany et al. "AddressSanitizer: a fast address sanity checker." In: *2012 USENIX Annual Technical Conference (USENIX ATC 2012)*. 2012, pp. 309–318.

[156] Du Shen et al. "Automating Performance Bottleneck Detection Using Search-based Application Profiling." In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. ACM, 2015, pp. 270–281. DOI: 10.1145/2771783.2771816. URL: http://doi.acm.org/10.1145/2771783.2771816.

[157] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. "Improving the resilience of an IDS against performance throttling attacks." In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2012, pp. 167–184.

[158] Govind Sreekar Shenoy, Jordi Tubella, and Antonio Gonz'lez. "Hardware/-Software Mechanisms for Protecting an IDS Against Algorithmic Complexity Attacks." In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE. 2012, pp. 1190–1196.

[159] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. "Recognizing Functions in Binaries with Neural Networks." In: *USENIX Security Symposium*. 2015, pp. 611–626.

[160] Matthew S Simpson and Rajeev K Barua. "MemSafe: ensuring the spatial and temporal memory safety of C at runtime." In: *Software: Practice and Experience* 43.1 (2013), pp. 93–128.

[161] Emin Gün Sirer and Brian N Bershad. "Using production grammars in software testing." In: *ACM SIGPLAN Notices*. Vol. 35. 1. ACM. 1999, pp. 1–13.

[162] Suphannee Sivakorn et al. "HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations." In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 521–538.

[163] Randy Smith, Cristian Estan, and Somesh Jha. "Backtracking algorithmic complexity attacks against a NIDS." In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 89–98.

[164] Linhai Song and Shan Lu. "Performance Diagnosis for Inefficient Loops." In: *Under Submission* ().

[165] Varun Srivastava et al. "A security policy oracle: Detecting security holes using multiple API implementations." In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 343–354.

[166] *Stack Exchange Network Status - Outage Postmortem - July 20, 2016*. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[167] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++." In: *Proceedings of the 13th Annual*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society. 2015, pp. 46–55.

[168] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2016.

[169] *Stuxnet - Wikipedia*. https://en.wikipedia.org/wiki/Stuxnet. (Accessed on 01-18-2018).

[170] Fang-Hsiang Su et al. "Code relatives: detecting similarly behaving software." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 702–714.

[171] Xiaoshan Sun, Liang Cheng, and Yang Zhang. "A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis." In: *Communications (ICC), 2011 IEEE International Conference on*. IEEE. 2011, pp. 1–5.

[172] Nikhil Swamy et al. "Secure distributed programming with value-dependent types." In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 266–278. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034811. URL: https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.

[173] *SWFTOOLS*. http://www.swftools.org/. (Accessed on 02-08-2018).

[174] *Telephone World - The Crash of the AT&T Network (1990)*. http://www.phworld.org/history/attcrash.htm. (Accessed on 01-18-2018).

[175] *Tesla driver killed while using autopilot was watching Harry Potter, witness says | Technology | The Guardian*. https://www.theguardian.com/technology/2016/jul/01/tesla-driver-killed-autopilot-self-driving-car-harry-potter. (Accessed on 01-18-2018).

[176] *The EFF SSL Observatory*. https://www.eff.org/observatory.

[177] Caroline Tice et al. “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.” In: *USENIX Security Symposium*. 2014, pp. 941–955.

[178] *Timeline of programming languages - Wikipedia*. https://en.wikipedia.org/wiki/Timeline_of_programming_languages. (Accessed on 01-29-2018).

[179] Tool Interface Standard. *The .xz File Format*. http://tukaani.org/xz/xz-file-format.txt. 2009.

[180] Nigel Tracey et al. “A search-based automated test-data generation framework for safety-critical systems.” In: *Systems engineering for business process change: new directions*. Springer, 2002, pp. 174–213.

[181] *Undefined Behavior Sanitizer - Clang 4.0 documentation*. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[182] *U.S. GAO - Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. https://www.gao.gov/products/IMTEC-92-26. (Accessed on 01-18-2018).

[183] Spandan Veggalam et al. “Ifuzzer: An evolutionary interpreter fuzzer using genetic programming.” In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 581–601.

[184] *VirusShare.com*. https://virusshare.com/.

[185] *Vulnerability distribution of cve security vulnerabilities by types*. https://www.cvedetails.com/vulnerabilities-by-types.php.

[186] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. “Synthesizing highly expressive sql queries from input-output examples.” In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2017, pp. 452–466.

[187] Shuai Wang, Pei Wang, and Dinghao Wu. “Semantics-aware machine learning for function recognition in binary code.” In: *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE. 2017, pp. 388–398.

[188] Shuai Wang and Dinghao Wu. "In-memory fuzzing for binary code similarity analysis." In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 319–330.

[189] Tielei Wang et al. "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2009.

[190] Xi Wang et al. "Improving integer security for systems with KINT." In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2012.

[191] Xi Wang et al. "Towards optimization-safe systems." In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[192] Zhimin Wang, Sebastian Elbaum, and David S Rosenblum. "Automated generation of context-aware tests." In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 406–415.

[193] *WashingtonPost.com: Cold War Report*. http://www.washingtonpost.com/wp-srv/inatl/longterm/coldwar/shatter021099b.htm. (Accessed on 01-18-2018).

[194] *'We Did Nothing Wrong'*. http://www.baselinemag.com/c/a/Projects-Processes/We-Did-Nothing-Wrong. (Accessed on 01-18-2018).

[195] Joachim Wegener and Matthias Grochtmann. "Verifying timing constraints of real-time systems by means of evolutionary testing." In: *Real-Time Systems* 15.3 (1998), pp. 275–298.

[196] *Why does Stack Overflow use a backtracking regex implementation? - Meta Stack Overflow*. https://meta.stackoverflow.com/questions/328376/why-does-stack-overflow-use-a-backtracking-regex-implementation.

[197] James R Wilcox et al. "Verdi: a framework for implementing and formally verifying distributed systems." In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 357–368.

[198] Valentin Wüstholz et al. “Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions.” In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 3–20.

[199] Weilin Xu, Yanjun Qi, and David Evans. “Automatically evading classifiers A Case Study on PDF Malware Classifiers.” In: *Proceedings of the 2016 Network and Distributed Systems Symposium (NDSS)*. 2016.

[200] Xiaojun Xu et al. “Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection.” In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 363–376.

[201] *XZ Utils*. http://tukaani.org/xz/. 2015.

[202] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers.” In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, California, USA: ACM, 2011, pp. 283–294.

[203] Zuoning Yin et al. “How do fixes become bugs?” In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 26–36.

[204] Suan Hsi Yong and Susan Horwitz. “Protecting C programs from attacks via invalid pointer dereferences.” In: *ACM SIGSOFT Software Engineering Notes*. Vol. 28. 5. ACM. 2003, pp. 307–316.

[205] Lian Yu et al. “Generating test cases for context-aware applications using bigraphs.” In: *Software Security and Reliability (SERE), 2014 Eighth International Conference on*. IEEE. 2014, pp. 137–146.

[206] Michal Zalewski. *american fuzzy lop*. http://lcamtuf.coredump.cx/afl/.

[207] Steve Zdancewic. “Challenges for information-flow security.” In: *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*. 2004, p. 6.

[208] Andreas Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: *Software Engineering-ESEC/FSE'99*. Springer. 1999, pp. 253–267.

[209] Chao Zhang et al. “IntPatch: Automatically Fix Integer-overflow-to-buffer-overflow Vulnerability at Compile-time.” In: *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*. 2010.