UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Fuzzing and Symbolic Execution to Identify and Patch Bugs

Permalink
https://escholarship.org/uc/item/7d87227q

Author
Salls, Christopher

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Santa Barbara

Fuzzing and Symbolic Execution to Identify and Patch Bugs

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Christopher Allen Salls

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Yan Shoshitaishvili

December 2020

The Dissertation of Christopher Allen Salls is approved.

Professor Yan Shoshitaishvili

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

December 2020

Fuzzing and Symbolic Execution to Identify and Patch Bugs

Copyright © 2020

by

Christopher Allen Salls

This dissertation is dedicated to my parents, who have been my inspiration throughout my entire life. I will always be amazed that my mom managed to finish her PhD while raising three kids.

Acknowledgements

A PhD is quite a long journey, and this one is no different. There have been so many ups and downs, moments where each day brought new excitement and success, and moments where disappointment took over. It is because of the people in my life, who were there to support me, teach me, and encourage me, that this thesis got finished.

First of all, I need to acknowledge the two people that made the biggest impact in shaping me, my parents. Throughout my school years, my mother (Jenny) and my father (Mitch) constantly encouraged me to learn as much as possible. They would never let me or my brothers settle for the bare minimum. Although at the time, I might have complained about the extra work, I know it was instrumental in getting me to where I am. My mother even gave me my first introduction into programming when she showed me that I could write programs for my new TI-84 calculator. I also have to mention my brothers, Kevin and Brian, both have been great supporters throughout my life, as well as a catalyst for my competitive personality which would later drive me through grad school.

In my undergrad at University of Nevada, I was able to experience academic research for the first time when one professor, Dr. Bebis, saw potential in me and offered to let me work for him on Computer Vision research. That experience got me excited to do research, and it was also when I first started considering a doctoral degree. Two other professors I especially want to recognize are Dr. Ramazan and Dr. Deaconu. They taught a weekly problem solving/math study, wherein I was able to hone my ability to approach difficult problems. This would later make a huge difference in my career as a PhD student.

Throughout my PhD at the Security Lab of UC Santa Barbara, I was surrounded by amazing people who would help guide me, challenge me, and continually push my limits. This was made possible by my professors Giovanni Vigna and Christopher Kruegel. I am so grateful that they took me on as a student when I had no experience in computer security. It is because

of their guidance, support, and the lab they created that I was able to excel during these last 6 years.

I'm thankful for all of the PhD students, interns, and teammates I worked with at the SecLab and on the Shellphish CTF team that made an impact on me in the last six years. Although, there are too many people to thank them all individually here, there are a few that I will list. Yan, a senior grad student who became my mentor early on in binary analysis research; I cannot express enough gratitude for his help guiding my research. Amat and Nick, two friends and teammates who propelled me into CTFs and binary exploitation. Without their help, I would not have been able to have nearly the level of success as I achieved during my graduate studies. Jake, a fellow student and a great friend, who would join me in vulnerability research in the lab and would go on to start a company with me; together we have truly achieved some great things.

Finally, I want to recognize Chani, one of the most determined and hard-working people I know. She worked hard to join the lab as a masters student, and soon after she would become my girlfriend. While in the lab, we had the opportunity to work together on two projects, one which was her thesis, and then one which would become my last paper. Later, her support would push me through the many sleepless nights to finish this thesis.

Curriculum Vitæ
Christopher Allen Salls

Education

2020	Ph.D. in Computer Science, University of California, Santa Barbara.
2013	B.S. in Computer Science, B.S. in Mathematics, University of Nevada, Reno.

Publications

1. Christopher Salls, Aravind Machiry, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna *Exploring Abstraction Functions in Fuzzing*. CNS 2020.
2. Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, Giovanni Vigna *Neurlux: dynamic malware analysis without feature engineering*. ACSAC 2019.
3. Yan Shoshitaishvili, Antonio Bianchi, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, Christopher Salls, Nick Stephens, Ruoyu Wang, Giovanni Vigna *Mechanical phish: Resilient autonomous hacking*. S&P 2018.
4. Christopher Salls, Aravind Machiry, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna *Piston: Uncooperative remote runtime patching*. ACSAC 2017.
5. Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, Giovanni Vigna *Difuze: Interface aware fuzzing for kernel drivers*. CCS 2017.
6. Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, Giovanni Vigna *Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance*. CCS 2017.
7. Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna *BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments*. NDSS 2017.
8. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, Giovanni Vigna *SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis*. S&P 2016.
9. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*. NDSS 2016.

Abstract

Fuzzing and Symbolic Execution to Identify and Patch Bugs

by

Christopher Allen Salls

Our computers, phones, and other smart devices are running a vast and ever increasing amount of software. This provides us with many capabilities that we make use of throughout our everyday lives. However, it also brings with it a large attack surface, wherein vulnerabilities could lie. A single vulnerability in any component could be maliciously exploited to gain access to private information or cause damage. This weakness is compounded by the imbalance between attackers and those who secure software; defenders must eliminate *all* of the vulnerabilities to have secure software, whereas an attacker may only need one vulnerability to be able to craft an exploit. Of course, all of this software cannot be audited, checked for bugs, and made secure manually; effective automated techniques are needed.

One of the most common causes of insecure software is memory corruption vulnerabilities. Memory corruption is frequently found in unsafe languages, such as C and C++, and can take many forms. Currently, the main methods of finding memory corruption are fuzzing and static analysis. However, both of these have major weaknesses that prevent finding many of the bugs present in modern software. The goal of my research is to improve upon the available techniques to make them more capable of finding bugs in real-world programs. To this end, the first work of my PhD identifies a key weakness in fuzzers—that they are impeded by difficult checks, such as string comparisons or magic numbers. With this in mind, we designed a new tool that combines fuzzing with symbolic execution, such that it can now solve for difficult checks and be able to continue fuzzing beyond them.

Of course, finding bugs is only one part of the problem. In my next work, I worked on a

novel application of symbolic execution to hot-patch vulnerable devices. Many IoT devices are not created with any built-in mechanism to update, and could be an easy target for attackers should a vulnerability be discovered. My research allows such devices to be automatically patched in event that a code execution vulnerability is found, without any support needed from the device itself.

In the final two projects of my PhD, I identify fundamental properties of fuzzers and use these to create new paradigms of fuzzing, which can find more vulnerabilities. One work formalizes how fuzzers find new inputs to mutate. Then, using this formalization, I design new strategies for fuzzing, which show improved capabilities for finding bugs. The last work I present here is a new technique for fuzzing JavaScript engines, called *Token-Level Fuzzing*. Instead of mutating individual bytes, Token-Level Fuzzing mutates entire tokens, replacing them with another valid token. Using this technique, we are able to find bugs which no other published fuzzer has found. I hope the techniques presented here can bring automated bug finding a step forward and be further built upon, as we try to eliminate memory corruption vulnerabilities.

In this thesis, I show that we can leverage the structure of binary programs and the essential properties of the data which they process to improve the effectiveness of vulnerability discovery based on fuzzing.

Contents

Curriculum Vitae **vii**

Abstract **viii**

1 Introduction **1**
1.1 Permissions and Attributions 5

2 Background on Vulnerability Discovery Techniques **6**
2.1 Static Analysis 7
2.2 Dynamic Analysis 8

3 Augmenting Fuzzing Through Selective Symbolic Execution **10**
3.1 Introduction 10
3.2 Driller Overview 14
3.3 Fuzzing 18
3.4 Selective Concolic Execution 22
3.5 Evaluation 31
3.6 Discussion 46
3.7 Conclusion 52

4 Uncooperative Remote Runtime Patching **54**
4.1 Introduction 54
4.2 Overview 58
4.3 Patch Generation 61
4.4 Repair Planning 64
4.5 Remote Patching 76
4.6 Evaluation 79
4.7 Limitations 83
4.8 Conclusion 85

5 **Exploring Abstraction Functions in Fuzzing** **86**
5.1 Introduction 86
5.2 Formalizing 88
5.3 Abstraction Functions Explored 96
5.4 Implementation 100
5.5 Evaluation 101
5.6 Discussion 110
5.7 Conclusion 110

6 **Token-Level Fuzzing** **112**
6.1 Introduction 112
6.2 Motivation 115
6.3 Overview 119
6.4 Evaluation 125
6.5 Discussion 136
6.6 Conclusion 138

7 **Related Work** **140**
7.1 Driller 140
7.2 Piston 143
7.3 Abstraction Functions in Fuzzing 146
7.4 Token-Level Fuzzing 148

8 **Looking Forward** **152**

Bibliography **154**

Chapter 1

Introduction

As technology has progressed, it has become more ingrained in our lives. We now have devices in our houses which listen to commands, turn lights on and off for us, and control the temperature. Many of us use technology to make purchases online, communicate with other people, access our banking information, transfer money, and more. Outside of personal use, software is used in many critical infrastructure applications such as power plants, satellites, and medical devices. Computing technology has truly become essential in our society, and, as such, software vulnerabilities are more serious now than ever before. If an attacker is able to break into someone's computer or smartphone, they can gain access to everything that it has access to. Even more seriously, if an attacker can target software that is part of the nation's critical infrastructure they could cause physical damage or disrupt society. As technology has become more ingrained in our society, it has become more paramount that it is secure.

A major source of insecurity is from languages, such as C and C++, that have no guarantee of memory safety. In these languages, a programming error can corrupt memory, resulting in undefined behavior. This corruption can in turn be used by a skilled attacker to control the program, allowing the attacker to run whatever code they wish. Although programming language enthusiasts might have hoped that safer languages would have taken over by now,

much of the most commonly used software is still written in C and C++. For example, every major browser uses large amounts of C/C++ code [1–4], as well as every major operating system [5–7]. This situation is not going to change anytime soon, so it is imperative that we find ways to secure code. One direction is, of course, advancements in mitigations. However, this is not a complete solution for many reasons. Many systems will not receive those mitigations anytime soon, and furthermore mitigations are frequently bypassed [8, 9]. One example is the iPhone's recently added Pointer Authentication Codes, which has been bypassed in many ways [10]. Although, this is a good direction, it is not enough.

I believe the most impactful direction is currently to improve automated bug finding, and, as such, this has been the primary focus of my research. Of course, automated bug finding is a very challenging problem; finding all the bugs in a program is as difficult as solving the halting problem [11]. The typical methods either use static techniques to identify buggy patterns, or to use dynamic techniques to try various inputs and see if they can trigger a crash. Dynamic techniques have more potential for purely automated analysis, because they produce a crashing input, and can continually run, getting deeper coverage in the target program.

In the Cyber Grand Challenge, the world's first automated hacking competition, the teams had to design a system which would automatically find and exploit bugs. When we attempted to apply popular fuzzers to the Cyber Grand Challenge, we got first-hand experience with a major limitation of fuzzing; this approach is hindered by difficult constraints, such as checksums, string comparisons, and magic values. This would become the inspiration for the first work contained in this thesis, *Driller*. Driller uses the insight that fuzzing, on its own, is quite dumb—it can only solve simple constraints. However, it is very fast at exploring code, triggering different cases, and maximizing coverage. Therefore, we can pair it up with a technique which can solve the difficult constraints, such as symbolic execution. We apply symbolic execution in a targeted manner while fuzzing, just to solve the constraints that fuzzing was not able to solve on its own. With this combination, Driller was able to find many more bugs than

fuzzing alone.

Of course, finding bugs is just one part of the problem; they have to be patched too. In most cases, patching a bug requires updating software, stopping the software, and restarting it [12]. However, this does not work for all software. There are critical systems that cannot be offline, even to update. For these situations, a technique called hotpatching [13] can be used to update the software while it is running. The major limitation of hotpatching is that it requires support; the device and its software have to be built with the capability to be hotpatched. Of course, especially for legacy code, this is not the case. There are many systems that have to wait for updates, and that would leave them unprotected, risking severe damage [14].

Furthermore, with the internet of things, there are many devices that cannot be updated at all [15]. There are vendors that are out of business and will not update old devices, as well as vendors that did not include the ability to update. What happens if a vulnerability is discovered on one of these systems? If the devices are used in people's homes, an attacker could use it to spy on them, or the attacker could simply use the compromised device as part of a large botnet [16]. Thus, we need a way to secure devices in this scenario.

Given the state of things, we realized that we needed a system that could hotpatch devices, without their cooperation. Fortunately, if there is a severe-enough vulnerability this can be done. Our idea was that by using the vulnerability we could hijack control of the insecure device and force a patch to be applied. This resulted in my next work, *Piston*. Given an exploit, Piston could automatically analyze the exploit and the patch, then apply it to a running device safely, protecting it from further exploitation.

In the years following the publication of Driller, there was a surge of research into fuzzing, exploring ways to fuzz faster [17], [18], advanced methods of mutating inputs [19], and different ways of choosing which inputs will get mutated [20, 21]. However, we noticed that there was a lot of ad hoc exploration, without any sort of grounding. In order to provide that grounding, we created a formalization of input selection in fuzzing, using methods from ab-

stract interpretation. We realized that input selection is similar to finding new elements in an abstract state space, and showed how existing fuzzing techniques all fit into the formalization.

Our formalization also led us to realize that most works only focus on a single abstraction for choosing which inputs will be fuzzed/mutated. We designed a very simple multi-abstraction fuzzer and found that it did better than a fuzzer based on any single abstraction. Our results show that fuzzing research has been missing an easy way to improve the performance of most evolutionary fuzzers, and we hope our formalization will provide a basis for further grounded research.

Up until this point, all the vulnerability research was applicable to any sort of target, from operating systems ot small test binaries, but now we wanted to tackle one of the hardest problems in fuzzing, interpreters. There are many high-value applications that use interpreters, one famous example being JavaScript engines. Vulnerabilities in JavaScript engines could potentially affect millions of users, and they are popular in targeted attacks [22]. Fuzzing interpreters is quite different from fuzzing other types of programs. One of the biggest differences is that interpreters take highly structured input in the form of a language. Most fuzzing techniques generate inputs which are too garbled to even parse and run. In order to fuzz interpreters effectively, I designed a completely new class of fuzzing, called *Token-Level Fuzzing*. Applying it to JavaScript engines, I show it finds more bugs than state-of-the-art JavaScript fuzzers.

One goal of my research is that I always try to approach a fundamental aspect of a problem in a unique way. I hope to continue this in my future work as well. I believe there are still major discoveries that will make a difference in how we look at software security, and that these will come from the people that can think outside of the box when approaching a problem. I hope that as I continue my research I can make an impact in helping the world have safer software and to further protect people from cyber threats.

1.1 Permissions and Attributions

1. The content of chapter 3 is the result of a collaboration with Nick Stephens, John Grosen, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2016 edition of the Network and Distributed System Security Symposium.

2. The content of chapter 4 is the result of a collaboration with Yan Shoshitaishvili, Nick Stephens, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2017 edition of the Annual Computer Security Applications Conference.

3. The content of chapter 5 is the result of a collaboration with Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2020 edition of the IEEE Conference on Communications and Network Security.

4. The content of chapter 6 is the result of a collaboration with Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna, and is currently in submission to the 2021 edition of the USENIX Security Symposium.

Chapter 2

Background on Vulnerability Discovery Techniques

In this chapter I will give an overview of the classes of modern vulnerability discovery techniques, and their advantages and disadvantages, There are many considerations that can go into such a discussion and it is important to remember the best is often to use a combination of the different analysis techniques. Because finding bugs is an NP-hard problem we have to make trade-offs, and each of the analysis techniques in this section will have different advantages and disadvantages. Some of the main considerations are:

- **Human effort.** Analysis techniques will require varying efforts from a human analysis. The effort could be as small as patching a clearly identified bug, to writing code to check for a specific bug. Techniques that can run with less human input are more suitable for continual bug discovery.

- **Replayable inputs.** If an analysis can output an input which will trigger a bug, it is known for certain that a bug exists. This can be quite helpful in enabling a developer to track down a particular bug. This also proves a bug is not a false positive.

- **Scalability.** Some analyses output their results and those cannot be improved with further compute time. On the other hand, some techniques can always run with more compute hours and increase their odds of finding bugs.
- **Root Causes.** The root cause of a vulnerability can be quite different from the crash that appears [23]. In these cases, understanding a crashing input may require a significant amount of effort to identify the real cause of the bug.

One consideration often missed in academic works is that more bugs is not always the goal of an analysis. The goal might be to find *different* vulnerabilities than other techniques. Vulnerability discovery should be a multi-pronged effort, wherein different techniques are used to try to find and eliminate as many bugs as possible, rather than using the one best technique. In fact, targeted automated bug discovery techniques have shown quite promising results.

2.1 Static Analysis

Static analyses attempt to find vulnerabilities in software without actually executing the target. This can be done on source code [24], on an intermediate language [25], or on the binary itself [26]. Some static techniques attempt to find patterns which are known to be buggy [27]. The simplest form of this may just look for functions that are known to be unsafe, whereas more complicated techniques might analyze how the data flows between functions to ensure that they are called in a safe manner.

Another common static analysis approach is to interpret a program over an abstract domain. That is an over-approximation is constructed for variables, memory locations etc, and the program is interpreted until a fixed point is reached. One such analysis is Value-Set Analysis, which attempts to identify a tight over-approximation of the values used in instructions as well as what regions of memory might be used in memory accesses [28]. VSA does this by

computing a *Value-set* which is a set that contains all the possible values of each register and memory value at all the program points. Then this analysis can be used to detect bugs, such as buffer overlaps, which could indicate a bug from a buffer overflow.

A major advantage of static analyses is that it analyze the entire code base; even in hard-to-reach areas of code, it can identify bugs. However, in order to analyze hard-to-reach code, it has to make assumptions or over-approximate the code, which may also lead to a large number of false positives. False-positives reduce confidence in the analysis and can cost the security engineers time. Another weakness is the lack of a testcase which can be used to reproduce the crash. Static analyses can only determine if there may be a bug at a particular point, not how to trigger it.

2.2 Dynamic Analysis

On the other hand, dynamic approaches actually run the target, either concretely or in an emulator. When used to find vulnerabilities, they typically execute the program from the entry point, allowing the analysis to produce a concrete input which triggers any vulnerabilities found. Executing from the entry point also comes with the benefit of having a more complete view of the program, often concrete values for what registers and memory regions may hold. However, executing from the beginning of a program also brings the challenge of needing to match the format that the target program expects to some degree of accuracy, otherwise only error paths will be triggered.

Lots of work in dynamic analysis vulnerability is finding techniques which can execute deeper parts of the code and cover more of the program. Fuzzing is the basic technique used most in industry. In fuzzing, random-sh inputs are generated and given to the program as inputs. in order to make fuzzing able to explore the state-space of a program effectively, most fuzzers rely on a technique called evolutionary fuzzing. By tracking how much coverage inputs

trigger in a program, the fuzzer can choose to keep around those that hit new basic blocks. These inputs will then be further mutated to try to iteratively explore deeper in an *evolutionary* manner. Two of the state-of-the-art evolutionary fuzzers which are used as the base of much of modern fuzzing research are AFL and Libfuzzer [29, 30]. One aspect of fuzzing that makes it very popular is its use of randomness, which allows it to continually be able to produce new results, and make effective use of a large number of CPU hours.

Another technique which has been popular both in research [31] and in industry [32] is symbolic execution. Symbolic execution involves using a special emulation environment to run the target program, where input is not concrete, rather it is given a symbolic value. As the target is executed, constraints are added onto the symbolic variables. Later, a constrain solver can be queried to determine possible values for all variables in the current program state. Symbolic execution is quite powerful in that it can determine correct values as it executes and easily handle many constraints on the input, but the downside is that it is very slow [33] and suffers from path explosion as too many paths become possible [31].

In my work, I chose to focus more on dynamic execution techniques, in particular fuzzing, with a bit of symbolic execution. I believe these techniques can make more effective use of the large amount of compute hours that companies have available. Fuzzing can run continuously, and can frequently find bugs that look nothing like any previous bug [34]. Furthermore, it can find bugs as they are introduced, and with a crashing input it is easy to pin down what commit is the one that introduced a vulnerability. Fuzzing has a lot of promise, and it still has a lot of room to grow as more advancements are made. I hope this work will make an impact in the efficacy of fuzzing and its ability to eliminate more security vulnerabilities in the software we use.

Chapter 3

Augmenting Fuzzing Through Selective Symbolic Execution

3.1 Introduction

Despite efforts to increase the resilience of software against security flaws, vulnerabilities in software are still commonplace. In fact, in recent years, the occurrence of security vulnerabilities has increased to an all-time high [35]. Furthermore, despite the introduction of memory corruption and execution redirection mitigation techniques, such software flaws account for over a third of all vulnerabilities discovered in the last year [36].

Whereas such vulnerabilities used to be exploited by independent hackers who wanted to push the limits of security and expose ineffective protections, the modern world has moved to nation states and cybercriminals using such vulnerabilities for strategic advantage or profit. Furthermore, with the rise of the *Internet of Things*, the number of devices that run potentially vulnerable software has skyrocketed, and vulnerabilities are increasingly being discovered in the software running these devices [37].

While many vulnerabilities are discovered by hand, manual analysis is not a scalable

method for vulnerability assessment. To keep up with the amount of software that must be vetted for vulnerabilities, an automated approach is required. In fact, DARPA has recently lent its support to this goal by sponsoring two efforts: VET, a program on developing techniques for the analysis of binary firmware, and the Cyber Grand Challenge (CGC), in which participants design and deploy automated vulnerability scanning engines that will compete against each other by exploiting binary software. DARPA has funded both VET and the Cyber Grand Challenge with millions of dollars in research funding and prize money, demonstrating the strong interest in developing a viable approach to automated binary analysis.

Naturally, security researchers have been actively designing automated vulnerability analysis systems. Many approaches exist, falling into three main categories: static, dynamic, and concolic analysis systems. These approaches have different advantages and disadvantages. Static analysis systems can provide provable guarantees – that is, a static analysis system can show, with certainty, that a given piece of binary code is secure. However, such systems have two fundamental drawbacks: they are imprecise, resulting in a large amount of false positives, and they cannot provide "actionable input" (i.e., an example of a specific input that can trigger a detected vulnerability). Dynamic analysis systems, such as "fuzzers", monitor the native execution of an application to identify flaws. When flaws are detected, these systems can provide actionable inputs to trigger them. However, these systems suffer from the need for "input test cases" to drive execution. Without an exhaustive set of test cases, which requires considerable manual effort to generate, the usability of such systems is limited. Finally, concolic execution engines utilize program interpretation and constraint solving techniques to generate inputs to explore the state space of the binary, in an attempt to reach and trigger vulnerabilities. However, because such systems are able to trigger a large number of paths in the binary (i.e., for a conditional branch, they often create an input that causes the branch to be taken and another that does not), they succumb to "path explosion", greatly limiting their scalability.

Because of these drawbacks, most bug-triggering input produced by modern automated

analysis systems represents "shallow" bugs in the software. In the case of fuzzers, this is because fuzzers randomly generate new inputs to an application and they likely fail to successfully pass through input-processing code. Concolic execution engines, on the other hand, are often able to recreate properly formatted input to pass through input processing code, but tend to succumb to path explosion, limiting the "depth" of code that they can analyze. Thus, flaws that lie in the deeper logic of an application tend to be missed by these tools, and are usually discovered through manual analysis by human experts [38–40].

The difference between the types of bugs that can be found by fuzzing and concolic execution can also be viewed in terms of the way in which an application processes user input. We propose two different categories of user input: *general* input, which has a wide range of valid values (e.g., the name of a user) and *specific* input, which has a limited set of valid values (e.g., the hash of the aforementioned name). An application's checks for particular values of specific input effectively split an application into *compartments*, separated by such checks. Fuzzing is proficient at exploring possible values of general input, *within* a compartment, but struggles to identify the precise values needed to satisfy checks on specific input and drive execution flow *between* compartments. On the other hand, selective concolic execution is proficient at determining the values that such specific checks require and, if the path explosion problem were solved, can push execution between compartments.

For example, consider an application that processes commands from the user: the application reads a command name from the user, compares it against a list of commands, and passes user-supplied parameters to the appropriate command handler. In this case, the complex check would be the comparison of the command name: a fuzzer randomly mutating input would have a very small chance of sending the correct input. On the other hand, a concolic execution engine would be well-suited for recovering the correct command name, but might suffer a path explosion in the parameter-processing code. Once the correct command name is determined, a fuzzer is better-suited for exploring the different command parameters that could be sent,

without encountering a path explosion.

We realized that this observation can be used to combine multiple analysis techniques, leveraging their strengths while mitigating their weaknesses. For example, a fuzzer can be used to explore the initial compartment of an application and, when it is unable to go further, a concolic execution engine can be leveraged to guide it to the next compartment. Once there, the fuzzer can take over again, exploring the possible inputs that can be provided to the new compartment. When the fuzzer stops making progress again, the concolic execution engine can resume and direct the analysis to the next compartment, and so on. By doing this repeatedly, execution is driven deeper and deeper into the program, limiting the path explosion inherent to concolic execution and ameliorating the incompleteness of dynamic analysis.

Guided by this intuition, we created a system, called *Driller*, that is a novel vulnerability excavation system combining a genetic input-mutating fuzzer with a selective concolic execution engine to identify deep bugs in binaries. Combining these two techniques allows Driller to function in a scalable way and bypass the requirement of input test cases. In this paper, we will describe the design and implementation of Driller and evaluate its performance on 126 applications released as part of the qualifying event of the DARPA Cyber Grand Challenge.

Driller is not the first work to combine different types of analyses. However, existing techniques either support very specific types of vulnerabilities (while Driller currently detects any vulnerability that can lead to a program crash) [41, 42], do not take full advantage of the capabilities offered by dynamic analysis (and, specifically, fuzzing) [43], or are affected by the path explosion problem [44–47]. We show that Driller identifies more vulnerabilities in these binaries than can be recovered separately by either fuzzing or concolic execution, and demonstrate the efficacy of our approach by discovering the same number of vulnerabilities, within the same amount of time, on the same dataset, as the winning team of the Cyber Grand Challenge qualifying event. Furthermore, we perform additional evaluations to show that this would not be possible without Driller's contribution (i.e., using traditional fuzzing or symbolic

execution approaches).

In summary, this paper makes the following contributions:

- We propose a new method to improve the effectiveness of fuzzing by leveraging selective concolic execution to reach deeper program code, while improving the scalability of concolic execution by using fuzzing to alleviate path explosion.
- We designed and implemented a tool, Driller, to demonstrate this approach.
- We demonstrate the effectiveness of Driller by identifying the same number of vulnerabilities, on the same dataset, as the winning team of the Cyber Grand Challenge qualifying event.

3.2 Driller Overview

A core intuition behind the design of Driller is that applications process two different classes of user input: *general* input, representing a wide range of values that can be valid, and *specific* input, representing input that must take on one of a select few possible values. Conceptually, an application's checks on the latter type of input split the application into *compartments*. Execution flow moves between compartments through checks against specific input, while, within a compartment, the application processes general input. This concept is explored in more depth in Section 3.5.7 in the context of an actual binary in our experimental dataset.

Driller functions by combining the speed of fuzzing with the input reasoning ability of concolic execution. This allows Driller to quickly explore portions of binaries that do not impose complex requirements on user input while also being able to handle, without the scalability issues of pure concolic execution, complex checks on specific input. In this paper, we de-

fine "complex" checks as those checks that are too specific to be satisfied by input from an input-mutating fuzzer.

Driller is composed of multiple components. Here, we will summarize these components and provide a high-level example of Driller's operation. In the rest of the paper, we will describe these components in depth.

Input test cases. Driller can operate without input test cases. However, the presence of such test cases can speed up the initial fuzzing step by pre-guiding the fuzzer toward certain compartments.

Fuzzing. When Driller is invoked, it begins by launching its fuzzing engine. The fuzzing engine explores the first compartment of the application until it reaches the first complex check on specific input. At this point, the fuzzing engine gets "stuck" and is unable to identify inputs to search new paths in the program.

Concolic execution. When the fuzzing engine gets stuck, Driller invokes its selective concolic execution component. This component analyzes the application, pre-constraining the user input with the unique inputs discovered by the prior fuzzing step to prevent a path explosion. After tracing the inputs discovered by the fuzzer, the concolic execution component utilizes its constraint-solving engine to identify inputs that would force execution down previously unexplored paths. If the fuzzing engine covered the previous compartments before getting stuck, these paths represent execution flows into new compartments.

Repeat. Once the concolic execution component identifies new inputs, they are passed back to the fuzzing component, which continues mutation on these inputs to fuzz the new compartments. Driller continues to cycle between fuzzing and concolic execution until a crashing input is discovered for the application.

Figure 3.1: The nodes initially found by the fuzzer.

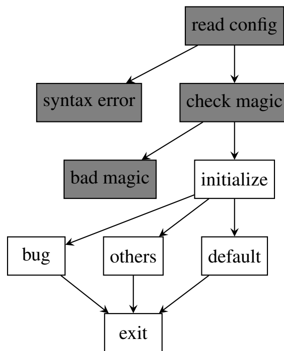


Figure 3.2: The nodes found by the first invocation of concolic execution.

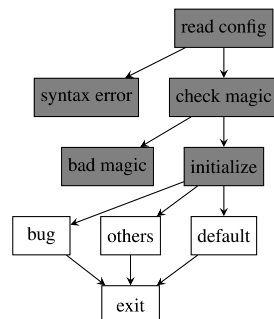


Figure 3.3: The nodes found by the fuzzer, supplemented with the result of the first Driller run.

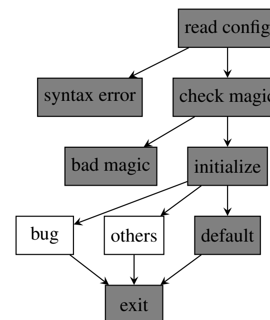
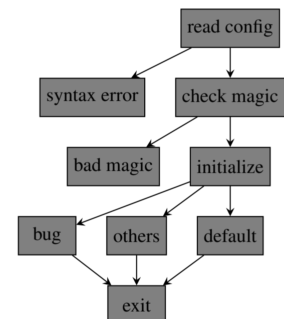


Figure 3.4: The nodes found by the second invocation of concolic execution.



3.2.1 Example

To elucidate the concept behind Driller, we provide an example in Listing 3.1. In this example, the application parses a configuration file, containing a magic number, received over an input stream. If the received data contains syntax errors or an incorrect magic number, the program exits. Otherwise, control flow switches based on input between a number of new compartments, some of which contain memory corruption flaws.

Driller begins its operation by invoking its fuzzing engine and fuzzing the first compartment of the application. These fuzzed nodes are shown shaded in a control-flow graph of the program in Figure 3.1. This fuzzing step explores the first compartment and gets stuck on the first complex check – the comparison with the magic number. Then, Driller executes the concolic execution engine to identify inputs that will drive execution past the check, into other program compartments. The extra transitions discovered by the concolic execution component, for this example, are shown in Figure 3.2.

After this, Driller enters its fuzzing stage again, fuzzing the second compartment (the initialization code and the check against keys in the configuration file). The coverage of the second

fuzzing stage is shown in Figure 3.3. As shown, the fuzzer cannot find any arms of the key switch besides the default. When this second fuzzing invocation gets stuck, Driller leverages its concolic execution engine to discover the "`crashstring`" and "`set_option`" inputs, shown in Figure 3.4. The former leads directly to the bug in the binary.

It is important to note that while neither symbolic execution nor fuzzing by themselves could find this bug, Driller can. There are several areas in this example where Driller's hybrid approach is needed. The parsing routines and initialization code have a great amount of complicated control flow reasoning about highly stateful data, which would lead to path explosion, slowing down symbolic execution to the point of uselessness. Additionally, and as noted before, the magic number check foils traditional fuzzing approaches by requiring highly specific input, too small to be reasonably found within its search space. Other common programming techniques that hinder fuzzing approaches include the use of hash functions to validate input. For this reason, a composition of concolic execution and fuzzing has the potential of achieving better results.

Listing 3.1: An example requiring fuzzing and concolic execution to work together.

```
1 int main(void) {
2         config_t *config = read_config();
3         if (config == NULL) {
4                 puts("Configuration syntax error");
5                 return 1;
6         }
7         if (config->magic != MAGIC_NUMBER) {
8                 puts("Bad magic number");
9                 return 2;
10        }
11        initialize(config);
12
13        char *directive = config->directives[0];
14        if (!strcmp(directive, "crashstring")) {
15                program_bug();
16        }
17        else if (!strcmp(directive, "set_option")) {
18                set_option(config->directives[1]);
19        }
20        else {
21                default();
22        }
23
24        return 0;
25 }
```

3.3 Fuzzing

Fuzzing is a technique that executes an application with a wide set of inputs, checking if these inputs cause the application to crash. To retain speed of execution, fuzzers are minimally invasive – they perform minimal instrumentation on the underlying application and mostly monitor it from the outside.

Recent years have seen many improvements to fuzzing engines. In this section, we will

detail improvements that are relevant to Driller's performance.

To implement Driller, we leveraged a popular off-the-shelf fuzzer, American Fuzzy Lop (AFL) [29]. Our improvements mostly deal with integrating the fuzzer with our concolic execution engine. No changes to the logic of AFL were made. AFL relies on instrumentation to make informed decisions on which paths are interesting. This instrumentation can be either introduced at compile-time or via a modified QEMU [48], we opted for a QEMU-backend to remove reliance on source code availability. While we discuss important features of Driller's fuzzer-component, AFL, in this section, we do not claim credit for their invention or implementation.

3.3.1 Fuzzer Features

A modern fuzzer implements many features to better identify crashing inputs. In this section, we will list and describe the most important AFL features, mentioning how they are used by Driller.

Genetic fuzzing. AFL carries out input generation through a genetic algorithm, mutating inputs according to genetics-inspired rules (transcription, insertion, etc.) and ranking them by a fitness function. For AFL, the fitness function is based on *unique* code coverage – that is, triggering an execution path that is different than the paths triggered by other inputs.

State transition tracking. AFL tracks the union of control flow transitions that it has seen from its inputs, as tuples of the source and destination basic blocks. Inputs are prioritized for "breeding" in the genetic algorithm based on their discovery of new control flow transitions, meaning that inputs that cause the application to execute in a different way get priority in the generation of future inputs.

Loop "bucketization". Handling loops is a complicated problem for fuzzing engines and concolic execution engines alike. To help reduce the size of the path space for loops, the following heuristic is performed. When AFL detects that a path contains iterations of a loop, a secondary calculation is triggered to determine whether that path should be eligible for breeding. AFL determines the number of loop iterations that were executed and compares it against previous inputs that caused a path to go through the same loop. These paths are all placed into "buckets" by the logarithm of their loop iteration count (i.e., 1, 2, 4, 8, and so on). One path from each bucket is considered for breeding in the genetic algorithm. This way, only $\log(N)$ paths must be considered for each loop as opposed to the naive approach of N paths.

Derandomization. Program randomization interferes with a genetic fuzzer's evaluation of inputs – an input that produces interesting paths under a given random seed might not do so under another. We pre-set AFL's QEMU backend to a specific random seed to ensure consistent execution. Later, when a crashing input is discovered, we use our concolic execution engine to recover any "challenge-response" behavior or vulnerabilities that rely on leaking randomness. For example, a "challenge-response" process in a binary echoes random data to the user and expects the same data echoed back to it. Without removing randomization, the fuzzing component would likely fail this check every time and explore very few paths. If the randomness is instead constant, the program accepts the same input each time, leaving the fuzzer (or the concolic execution component) free to find this one value and subsequently explore further. After a crash is found, the randomness can instead be modeled symbolically, as described in section 3.4.4, and the crashing input can be patched accordingly.

These features allow AFL to rapidly discover unique paths through an application, performing the brunt of the path discovery work within a given compartment of the application.

However, the limitations of fuzzing are well-known.

3.3.2 Fuzzer Limitations

Because fuzzers randomly mutate input, and genetic fuzzers, in turn, mutate input that has, in the past, generated unique paths through a binary, they are able to quickly discover different paths that process "general" input (i.e., input that has many different values that can trigger meaningful program behavior). However, the generation of "specific" input to pass complex checks in the application (i.e., checks that require inputs with one of very few specific values) is very challenging for fuzzers.

Consider the example in Listing 3.2.

Listing 3.2: A difficult program to fuzz.

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

This application reads a value from the user and compares it against a specific value. If the correct value is provided, the application will crash. However, due to the nature of fuzzing, it is extremely unlikely that a fuzzer will ever satisfy the predicate. For a non-instrumented fuzzer (i.e., one that chooses random values for the input), the likelihood that the fuzzer will discover the bug is the infinitesimal 1 out of 2^{32}. For an instrumented fuzzer, the control flow layout of this binary will result in a single path being discovered. Without the ability to prioritize new paths (as there are none), an instrumented fuzzer will be reduced to applying random mutations on the existing paths which is, in essence, the same as the non-instrumented case, with the same infinitesimally small chance of success.

3.3.3 Transition to Concolic Execution

Driller aims to complement the fundamental weakness of fuzzing, determining specific user input required to pass complex checks, by leveraging the strength of concolic execution. When the fuzzing component has gone through a predetermined amount (proportional to the input length) of mutations without identifying new state transitions, we consider it "stuck". Driller then retrieves the inputs that the fuzzer has deemed "interesting" in the current compartment and invokes the concolic execution engine on them.

The fuzzer identifies inputs as interesting if one of two conditions holds:

1. The path that the input causes the application to take was the first to trigger some state transition.

2. The path that the input causes the application to take was the first to be placed into a unique "loop bucket".

These conditions keep the number of inputs that are handed to the concolic execution component down to a reasonable number, while retaining a high chance of passing along inputs that the concolic execution can mutate to reach the next compartment in the application.

3.4 Selective Concolic Execution

When Driller determines that the fuzzer is unable to find additional state transitions, the concolic execution engine is invoked. The insight behind Driller's use of concolic execution is as follows: one of the main causes of fuzzers failing to find new state transitions in a program is the inability of fuzzers to generate specific input to satisfy complex checks in the code. The concolic execution engine is used to leverage a symbolic solver to *mutate* existing inputs that reach but fail to satisfy complex checks into new inputs that reach and *satisfy* such checks.

When Driller invokes the concolic execution engine, it passes all of the "interesting" inputs (as defined in Section 3.3.3) that were identified by the fuzzing engine. Each input is traced, symbolically, to identify state transitions that the fuzzing engine was unable to satisfy. When such a transition is identified, the concolic execution engine produces input that would drive execution through this state transition.

After the concolic execution engine finishes processing the provided inputs, its results are fed back into the fuzzing engine's queue and control is passed back to the fuzzing engine, so that it can quickly explore the newly found compartments of the application.

The remainder of this section will describe Driller's implementation of concolic execution and the specific adaptations that we made for Driller's problem domain.

3.4.1 Concolic Execution

We leveraged angr [37], a recently open-sourced symbolic execution engine, for Driller's concolic execution engine. The engine is based on the model popularized and refined by Mayhem and S2E [46, 47]. First the engine translates binary code into Valgrind's VEX [49] intermediate representation, which is interpreted to determine the effects of program code on a *symbolic state*. This symbolic state uses *symbolic variables* to represent input that can come from the user or other data that is not constant, such as data from the environment. A symbolic variable is a variable (such as X) that can yield a number of possible concrete solutions (such as the number 5). Other values, such as constants hardcoded in the program, are modeled as concrete values. As the execution progresses, *symbolic constraints* are added to these variables. A constraint is a limiting statement on the potential solutions of the symbolic value (for example, $X < 100$). A concrete solution is any value of X that will satisfy these constraints.

The analysis engine tracks all concrete and symbolic values in memory and registers (the aforementioned symbolic state) throughout execution. At any point in the program that the

engine reaches, a constraint resolution can be performed to determine a possible input that satisfies the constraints on all symbolic variables in the state. Such an input, when passed to a normal execution of the application, would drive the application to that point. The advantage of concolic execution is that it can explore and find inputs for any path that the constraint solver can satisfy. This makes it useful for identifying solutions to complex comparisons (up to and including certain hash functions) that a fuzzer would be unlikely to ever brute force.

Driller's symbolic memory model can store both concrete and symbolic values. It uses an index-based memory model in which read addresses may be symbolic, but write address are always concretized. This approach, popularized by Mayhem, is an important optimization to keep the analysis feasible: if both read and write addresses were symbolic, a repeated read and write using the same symbolic index would result in a quadratic increase in symbolic constraints or, depending on the implementation details of the symbolic execution engine, the complexity of the stored symbolic expressions. Thus, symbolic write addresses are always concretized to a single valid solution. Under certain conditions, as proposed by literature in the field, symbolic values are concretized to a single potential solution [47].

The symbolic memory optimizations increase the scalability of the concolic execution engine, but can result in an incomplete state space, where fewer solutions are possible. Unfortunately, this is a trade-off that must be made to make analysis of real-world binaries realistic.

3.4.2 Example

Concolic execution is good at solving different problems than fuzzing. Recall the example demonstrating the drawback of fuzzing, from Section 3.3.2, reproduced in Listing 3.3.. Because of the exactness of the input required to pass the check guarding the call to the `vulnerable` function, fuzzing is unable to explore that piece of code in a reasonable time frame.

Listing 3.3: A program that yields to concolic execution.

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

However, a concolic execution engine will be able to easily satisfy this check and trigger the `vulnerable` function. For this example, concolic execution only needs to explore a small number of paths to find one which reaches the bug in this example, but for bigger binaries and real-world examples, there will be far too many paths to explore in the same manner.

3.4.3 Limitations

The traditional approach to concolic execution involves beginning concolic execution from the beginning of a program and exploring the path state with the symbolic execution engine to find as many bugs as possible. However, this approach suffers from two major limitations.

First, concolic execution is slow. This is caused by the need to interpret application code (as opposed to natively executing it, as with a fuzzer) and by the overhead involved in the constraint solving step. Specifically, the latter operation involves the solution of an NP-complete problem, making the generation of potential inputs (and the determination of which conditional jumps are feasible) time-consuming.

Worse, symbolic execution suffers from the state explosion problem. The number of paths grows exponentially as the concolic execution engine explores the program, and it quickly becomes infeasible to explore more than a tiny fraction of the paths. Consider the example in Listing 3.4. In this program, the `vulnerable()` is triggered when the user enters exactly

25 `B` characters, but this is a condition difficult to express in a symbolic execution framework. Symbolic execution of this program will cause a huge state explosion as the simulated CPU steps down the recursive calls into the `check()` function. Each execution of the ternary conditional comparing a character to the literal `B` splits every simulated state into two, eventually resulting in 2^{100} possible states, which is an infeasible amount to process.

A genetic fuzzer that selects inputs based on state transitions, on the other hand, does not reason about the whole state-space of a program, but only on the state transitions triggered by inputs. That is, it will focus chiefly on the number of times, for example, the check on line 5 succeeds. That is, regardless of *where* the `B` characters are in the input, states will be judged based on the *number* of them in the input, avoiding the path explosion problem.

While progress has been made toward reducing this problem with intelligent state merging [50], the general problem remains.

Listing 3.4: A program that causes a path explosion under concolic execution.

```
1  int check(char *x, int depth) {
2          if (depth >= 100) {
3                  return 0;
4          } else {
5                  int count = (*x == 'B') ? 1 : 0;
6                  count += check(x+1, depth+1);
7                  return count;
8          }
9  }
10
11 int main(void) {
12         char x[100];
13         read(0, x, 100);
14
15         if (check(x, 0) == 25)
16                 vulnerable();
17 }
```

3.4.4 Concolic Execution in Driller

In most cases, fuzzing can adequately explore a large portion of paths on its own, simply by finding them with random bit flips and other mutation strategies. By utilizing native execution, it will outperform concolic execution in most cases where it can randomly trigger the paths. Thus, most of the work is offloaded from the concolic execution engine to the fuzzer, which will find many paths quickly, letting the concolic engine just work on solving the harder constraints.

When fuzzing is unable to discover inputs that result in new execution paths, the concolic execution engine is invoked. It traces the paths discovered by the fuzzing, identifies inputs that diverge into new program components, and performs limited symbolic exploration. Additionally, when a crashing input is found by the fuzzing component, the concolic execution engine "re-randomizes" it to recover the parts of a crashing input that are dependent on randomness and other environmental factors.

Pre-constrained Tracing

Driller uses concolic execution to trace the interesting paths from the fuzzer and generate new inputs. A key factor in the effectiveness of this approach is that it allows Driller to avoid the path explosion inherent in concolic exploration, because only the path representing the application's processing of that input is analyzed.

When traces are passed from the fuzzer to the symbolic execution, the goal is to discover new transitions that fuzzing had not previously found. Driller's concolic execution engine traces the input, following the same path that was taken by the fuzzer. When Driller comes upon a conditional control flow transfer, it checks if *inverting* that condition would result in the discovery of a new state transition. If it will, Driller produces an example input that will drive execution through the new state transition instead of the original control flow. By doing this Driller's concolic execution engine guides the fuzzing engine to new compartments of the

application. After producing the input, Driller continues following the matching path to find additional new state transitions.

Input Preconstraining

Driller uses *preconstraining* to ensure that the results of the concolic execution engine are identical to those in the native execution while maintaining the ability to discover new state transitions. In preconstrained execution, each byte of input is constrained to match each actual byte that was output by the fuzzer, e.g., `/dev/stdin[0] == 'A'`. When new possible basic block transitions are discovered, the preconstraining is briefly removed, allowing Driller to solve for an input that would deviate into that state transition. Preconstraining is necessary to generate identical traces in the symbolic execution engine and make the limited concolic exploration feasible.

To demonstrate how input preconstraining works in Driller, we use the example in Listing 3.5, which is similar to the example from Section 3.4.3 with the addition that, to reach the vulnerable function, we must provide a magic number (0x42d614f8) at line 18. After fuzzing the input, Driller eventually recognizes that it is not discovering any new state transitions, since the fuzzer alone cannot guess the correct value. When concolic execution is invoked to trace an input, Driller first constrains all of the bytes in the symbolic input to match those of the traced input. As the program is symbolically executed, there is only one possibility for each branch, so exactly one path is followed. This prevents the path explosion that was described in Section 3.4.3. When execution reaches line 18, however, Driller recognizes that there is an alternate state transition that has never been taken during fuzzing. Driller then removes the preconstraints that were added at the beginning of the execution not including the predicates placed by symbolically executing the program with the traced input. The bytes in the character array `x` are partially constrained by the path, and the value of `magic` is constrained by the equality check `if (magic == 0x42d614f8)`. The concolic execution engine thus cre-

ates an input that contains 25 instances of `B` and a `magic` value of 0x42d614f8. This passes the check in line 18 and reaches the vulnerable function.

Listing 3.5: An application showcasing the need for pre-constraining of symbolic input.

```
1 int check(char *x, int depth) {
2         if (depth >= 100) {
3                 return 0;
4         } else {
5                 int count = (*x == 'B') ? 1 : 0;
6                 count += check(x+1, depth+1);
7                 return count;
8         }
9 }
10
11 int main(void) {
12         char x[100];
13         int magic;
14         read(0, x, 100);
15         read(0, &magic, 4);
16
17         if (check(x, 0) == 25)
18                 if (magic == 0x42d614f8)
19                         vulnerable();
20 }
```

Limited Symbolic Exploration

In an attempt to reduce the number of expensive concolic engine invocations we also introduce a symbolic exploration stub to discover more state transitions lying directly after a newly discovered state transition. This symbolic exploration stub explores the surrounding area of the state transition until a configurable number of basic blocks has been traversed by the explorer. Once this number of blocks has been discovered, Driller concretizes inputs for all paths discovered by the explorer. We reason that doing this prevents the fuzzer from getting "stuck" quickly

after being provided with a Driller-generated input. In a number of cases, Driller generates a new input that gets only partway through a multi-part complex check and must immediately be retraced to allow the fuzzer to proceed deeper into the binary. The symbolic exploration stub is a small optimization which allows Driller to find further state transitions, before they are requested, without having to retrace its steps.

Re-randomization

Random values introduced during a program run can disrupt fuzzing attempts as described earlier. Listing 3.6 displays a small program which challenges the user to reflect back a random input. This makes fuzzing unstable because we can never know the concrete value of `challenge` without monitoring the program output.

Listing 3.6: A program which requires re-introducing randomness.

```
1 int main(void) {
2     int challenge;
3     int response;
4
5     challenge = random();
6
7     write(1, &challenge, sizeof(challenge));
8     read(0, &response, sizeof(response));
9     if (challenge == response)
10        abort();
11
12 }
```

Once a vulnerability is discovered, we use symbolic execution to trace crashing inputs and recover input bytes that need to satisfy dynamic checks posed by the target binary (such as the challenge-response in the example of Listing 3.6). By inspecting the symbolic state at crash time and finding the relationships between the application's output and the crashing input, Driller can determine the application's challenge-response protocol. In this example, we can

see that the symbolic bytes introduced by the call to `read` are constrained to being equal to the bytes written out by the call to `write`. After determining these relationships, we can generate an exploit specification that handles randomness as it occurs in a real environment.

3.5 Evaluation

To determine the effectiveness of our approach, we performed an evaluation on a large dataset of binaries. The goal of our evaluation is to show two things: first, Driller considerably expands the code coverage achieved by an unaided fuzzer, and, second, this increased coverage leads to an increased number of discovered vulnerabilities.

3.5.1 Dataset

We evaluated Driller on applications from the qualifying event of the DARPA Cyber Grand Challenge (CGC) [51], a competition designed to "test the abilities of a new generation of fully automated cyber defense systems" [51]. During the event, competitors had 24 hours to autonomously find memory corruption vulnerabilities and demonstrate proof by providing an input specification that, when processed by the application in question, causes a crash. There are 131 services in the CGC Qualifying Event dataset, but 5 of these involve communication between multiple binaries. As such functionality is out of scope for this paper, we only consider the 126 single-binary applications, leaving multi-binary applications to future work.

These 126 applications contain a wide range of obstacles that make binary analysis difficult, such as complex protocols and large input spaces. They are specifically created to stress the capabilities of program analysis techniques, and are not simply toy applications for hacking entertainment (unlike what is generally seen at Capture The Flag hacking competitions [52]). The variety and depth of these binaries allow for extensive testing of advanced vulnerability excavation systems, such as Driller. Furthermore, the results of the top competitors are avail-

able online, providing a litmus test for checking the performance of analysis systems against verified results.

3.5.2 Experiment Setup

We ran our experiments on a computer cluster of modern AMD64 processors. Each binary had four dedicated fuzzer nodes and, when the fuzzer requires concolic execution assistance, it sent jobs to a pool of 64 concolic execution nodes, shared among all binaries. Due to constraints on the available memory, we limited each concolic execution job to 4 gigabytes of RAM. In all of our tests, we analyze a single binary for at most 24 hours, which is the same amount of time that was given to the teams for the CGC qualifying event. We analyzed each binary until either a crash was found or the 24 hours had passed.

All crashes were collected and replayed using the challenge binary testing tools to verify that the reported crashes were repeatable in the actual CGC environment. Thus, these results are *real*, verified, and comparable to the actual results from the competition.

3.5.3 Experiments

We ran a total of three experiments in our evaluation. First, to evaluate Driller against the baseline performance of existing techniques, we attempted vulnerability excavation with a pure symbolic execution engine and a pure fuzzer. Then, we evaluated Driller on the same dataset.

The experiments were set up as follows:

Basic fuzzing. In this test, each binary was assigned 4 cores for fuzzing by AFL, but the concolic execution nodes were deactivated. The fuzzer had no assistance when it was unable to discover new paths. Note that changes were made to AFL's QEMU backend to improve performance on CGC binaries, however, as mentioned previously no core changes to AFL's logic were made.

Symbolic execution. We used an existing symbolic execution engine, based heavily on the ideas proposed by Mayhem [47], for the concolic execution test. To ensure a fair test against the state of the art, advanced state merging techniques were used to help limit the effects of state explosion, as proposed in Veritesting [50].

We analyze each binary by symbolically exploring the state space, starting from the entry point, checking for memory corruption. When a state explosion did occur, we used heuristics to prioritize paths that explored deeper into the application to maximize code coverage.

Driller. When testing Driller, each binary was assigned 4 cores for the fuzzing engine, with a total of 64 cores for the concolic execution component. The concolic execution pool processed symbolic execution jobs in a first-in-first-out queue as traces were requested by the fuzzing nodes when Driller determined that the fuzzers were “stuck”. Symbolic execution traces were restricted to a one-hour period and a 4 gigabyte memory limit to avoid resource exhaustion from analyzing large traces.

We will discuss several different facets of our evaluation of Driller. We will start by discussing the results of the three experiments in terms of Driller’s contribution to the number of vulnerabilities that we were able to find in the dataset. Next, we will discuss Driller’s contribution in terms of code coverage over existing techniques. Finally, we will focus on an example application from the CGC dataset for an in-depth case study to discuss how Driller increased code coverage and identified the vulnerability in that application.

3.5.4 Vulnerabilities

In this subsection, we will discuss the number of vulnerabilities that were discovered by the three experiments, and frame Driller’s contribution in this regard.

The symbolic execution baseline experiment faired poorly on this dataset. Out of the 126 applications, symbolic execution discovered vulnerabilities in only 16.

Out of the 126 Cyber Grand Challenge applications in our experimental dataset, fuzzing proved to be sufficient to discover crashes in 68. Of the remaining 58 binaries, 41 became “stuck” (i.e., AFL was unable to identify any new “interesting” paths, as discussed in Section 3.3, and had to resort to random input mutation) and 17, despite continuing to find new interesting inputs, never identified a crash.

In Driller’s run, the fuzzer invoked the concolic execution component on the 41 binaries that became “stuck”. Figure 3.7 shows the number of times that concolic execution was invoked for these binaries. Of these, Driller’s concolic execution was able to generate a total of 101 new inputs for 13 of these applications. Utilizing these extra inputs, AFL was able to recover an additional 9 crashes, bringing the total identified crashes during the Driller experiment to 77, meaning that Driller achieves a 12% improvement over baseline fuzzing in relation to discovered vulnerabilities.

Of course, most of the applications for which crashes were discovered in the Driller experiment were found with the baseline fuzzer. In terms of unique crashes identified by the different approaches, the fuzzer baseline discovered 55 crashes symbolic execution failed to discover. 13 of its vulnerabilities were shared with the symbolic execution baseline. A further 3 symbolic execution baseline vulnerabilities overlap with vulnerabilities recovered by Driller, leaving application for which the symbolic execution baseline alone found a vulnerability, and leaving 6 applications for which Driller’s approach was the only one to find the vulnerability. Essentially, Driller effectively merges and expands on the capabilities offered by baseline fuzzing and baseline concolic execution, achieving more results than both do individually. These results are presented in Figure 3.5.

In total, Driller was able to identify crashes in 77 unique applications, an improvement of 6 crashes (8.4%) over the union of the baseline experiments. This is the same number of crashes

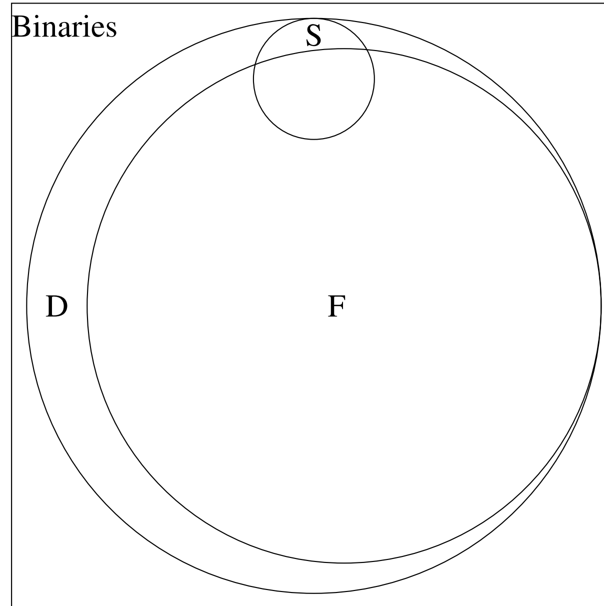
as identified by the top-scoring team in the competition (and significantly higher than any of the other competitors), in the same amount of time. Without Driller (i.e., with the two baseline approaches), we would not have achieved these results. Note that we are well-aware that the comparison to a participating team is only indicative and it is not meant to be qualitative. The participating team was operating under strict time constraints, with little or no space for errors. Our experiments benefit from additional time to prepare and our techniques could be refined throughout the course of Driller's development.

These results demonstrate that enhancing a fuzzer with selective concolic execution improves its performance in finding crashes. By advancing the state of the art in vulnerability excavation, Driller is able to crash more applications than the union of those found by fuzzing and by symbolic execution separately. While a contribution of 6 unique vulnerabilities might seem low compared to the total number of applications in the CGC qualifying event, these crashes represent vulnerabilities deep in their respective binaries, many of which require multiple concolic execution invocations to penetrate through several compartments.

3.5.5 State Transition Coverage

Selective symbolic execution is able to overcome a fundamental weakness of fuzzers when dealing with "magic" constants and other complex input checks. That means that, after the fuzzer is unable to identify new interesting inputs (for example, due to a failure to guess a hash or a magic number), the concolic execution engine can generate an input allowing the fuzzer to continue exploring paths beyond where it had become stuck. This aspect of Driller can be observed in Table 3.1, which shows the breakdown of how state transitions were found during execution. In applications in which the symbolic execution was able to find a new path, fuzzing alone had only found an average of 28.5% of the block transitions.

As expected, the symbolic traces account for only a small amount of new state transitions



Method	Crashes Found
Fuzzing	68
Fuzzing ∩ Driller	68
Fuzzing ∩ Symbolic	13
Symbolic	16
Symbolic ∩ Driller	16
Driller	77

Figure 3.5: The makeup of the experimentation results. The Venn Diagram shows the relative coverage of Basic Fuzzing (AFL), Symbolic Execution, and Driller in terms of finding crashes in the CGC dataset. The circle labeled **F** represents crashes found by fuzzing, **S** represents crashes found by symbolic execution, and **D** represents crashes found by driller. The table presents these results in terms of the relative effectiveness of the different methods and their improvement relative to each other. The attentive reader can see that Driller identifies a superset of the crashes found by Fuzzing and Symbolic Execution.

in these binaries (about 15.1% on average), as the symbolic exploration is limited in scope and reserved mostly for identifying and passing interesting checks. However, the inputs produced by the concolic execution engine help the fuzzing engine in successfully penetrating these state transitions. The fuzzing engine's subsequent modifications of these inputs allow it to find, on average, an additional 56.5% of state transitions. In total, for the applications in which the fuzzer eventually gets stuck and symbolic execution found a new path, 71.6% of the state transitions resulted from the inputs based on those that were generated during symbolic traces. The fact that the small numbers of concolically-contributed inputs result in a much larger set of state transitions that the fuzzer can explore demonstrates that the inputs generated by Driller's concolic execution engine stimulated a much deeper exploration of the application. It is important to keep in mind that this number only applies to 13 of the 41 applications which became ”stuck” and were able to have a new path identified by symbolic execution. These percentages are normalized over the total amount of basic blocks that we saw over the course of the experiment, as generating a complete Control Flow Graph statically requires heavyweight static analysis that is outside of the scope of this paper.

As discussed in Section 5.2, we consider a state transition to be an ordered pair of basic blocks (A,B) where block B is executed immediately following block A. In other words, a state transition is an *edge* in a Control Flow Graph where each node represents a basic block in the program. It is clear that if we find every state transition that we have complete code coverage. Similarly, if we find few state transitions, than we likely have very low coverage. Thus, it is reasonable to use the number of unique state transitions as a measure of code coverage. In Figure 3.6, we show how Driller improved the basic block coverage over time, by showing how many additional basic blocks were discovered as a result of Driller, that the fuzzer was unable to find on its own.

Type of State Transition	Percentage of discovered blocks across all binaries	Percentage of discovered blocks across binaries where concolic execution found at least one input
Initial Fuzzing Run	84.2	28.4
Identified by Concolic Execution	3.3	15.1
Post-Concolic Fuzzing Runs	12.5	56.5
Total	100	100

Table 3.1: Breakdown of what percentage of discovered state transitions were found by what method, among binaries which invoked concolic execution and binaries for which concolic execution identified at least one input.

3.5.6 Application Component Coverage

A goal of the symbolic traces in Driller is to enable the fuzzer to explore the various compartments in a binary, where the compartments may be separated by complex checks on user input. We expect to see inputs generated by invocations of the concolic tracer correspond to finding new compartments in the application. That is, the inputs generated by the concolic execution engine should enable the fuzzer to reach and explore new areas of code.

As shown in Figure 3.5, 68 of the 126 applications in the data set did not have any difficult checks that needed Driller's symbolic execution. These correspond to applications for which the fuzzing component independently found crashing inputs or for which it never became "stuck". These applications tend to be the ones with simple protocols and fewer complex checks. On the other hand, Driller was able to satisfy at least one difficult check in 13 of the binaries and multiple difficult checks in 4 of the binaries. These compartments are difficult for basic fuzzers to enter because of the specific checks separating them, but solvable by the hybrid approach employed by Driller.

Each invocation of concolic execution has the potential to guide execution to a new compartment in the application. This can be measured by analyzing the basic block coverage of Driller before a fuzzing round gets "stuck" and invokes concolic execution versus the coverage achieved by the subsequent round of fuzzing, after the concolic execution component pushed

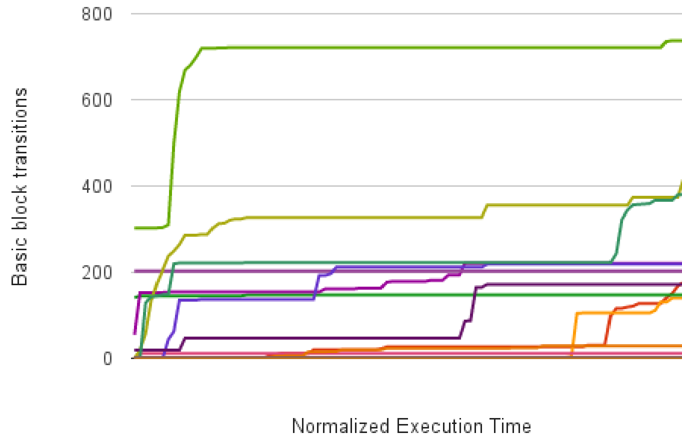


Figure 3.6: The number of additional basic blocks found by Driller over time, that the fuzzer was unable to find on its own. Execution time is shown normalized to the execution time of the binary, which varies depending on if/when it crashed. This graph includes the 13 binaries that invoked, and benefited from, concolic execution.

execution through to the next compartment. We present this in Figure 3.8, by showing the fraction of basic blocks, normalized to the total number of basic blocks discovered throughout the experiment, for each binary on which concolic execution was invoked, at each stage of the analysis. The graph demonstrates that Driller *does* drive execution into new compartments in applications, allowing the fuzzer to quickly explore a greater amount of code. We present an in-depth example in this for our case study in Section 3.5.7.

3.5.7 Case Study

This section will focus on a single application to explain, in-depth, Driller's operation. We will focus on the CGC qualifying event application whose identifier is `2b03cf01`. The interested reader can find the source code for this application on DARPA's github repository [53] under the public name `NRFIN_00017`. Additionally, we present the call graph of this binary, which we will refer to throughout this case study, in Figure 3.10. This graph demonstrates the

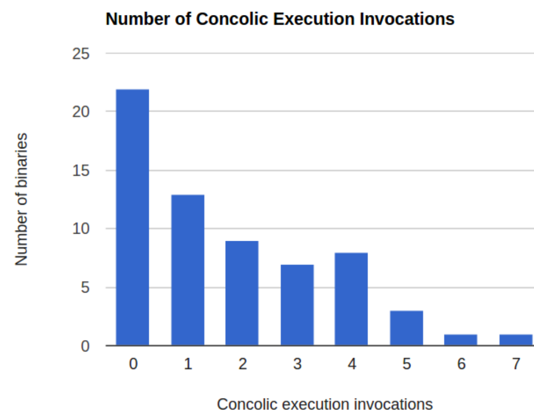


Figure 3.7: Graph showing how many times concolic execution was invoked in binaries where fuzzing could not crash the binary on its own.

performance of successive invocations of Driller's fuzzing and concolic execution components – the nodes discovered by successive fuzzing invocations are drawn in progressively darker colors and the transitions recovered by the concolic execution component are illustrated with differently-drawn edges. The different colors of nodes represent different compartments in this binary – within the compartment, fuzzing successfully produces inputs to trigger new interesting paths while concolic execution is needed to satisfy the complex checks and guide execution flow between compartments.

This application represents a power testing module, in which the client provides the server an electrical design and the server builds a model of the electrical connectivity. This is not a simple binary: it provides a variety of complex functionality to the user and requires properly formatted input, against which there are a number of complex checks.

When Driller fuzzes this binary, the first of these complex checks causes the fuzzer to get stuck almost immediately after finding only 58 basic blocks across a fairly small compartment of the application, consisting of a handful of functions containing initialization code. The fuzzing engine gets stuck on a check on user input. For convenience, the snippet in question, corresponding to node "A" in Figure 3.10, is reproduced in Listing 3.7, although, of course,

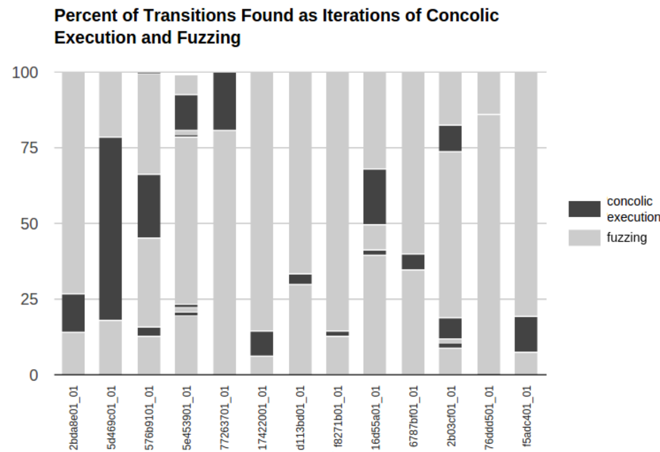


Figure 3.8: Graph showing how each invocation of concolic execution lead to more basic block transitions found. Only shown for binaries in which symbolic execution identified additional inputs.

Driller operates directly on binary code.

Looking at the source code, we see that the two primary commands called from the main loop require the user to give a specific 32-bit number to select a “mode of operation”. To call the function *do_build()*, the user must provide the number 13980, and to call the function *do_examine()*, the user must provide the number 809110. Although, these checks appear simple to a human, a fuzzer must essentially brute force them. Thus, the chance that the fuzzer will guess these magic numbers is minuscule, and, as a result, the fuzzing component gets stuck.

After the fuzzer is unable to identify new interesting paths, Driller invokes the concolic execution component to trace the inputs that the fuzzer has collected thus far, and find new state transitions. Driller finds inputs which will drive execution to both of the aforementioned functions, and returns them to the fuzzer for exploration. Again, the fuzzer gets stuck fairly quickly, this time at node “B” in Figure 3.10 at another complex check. Driller’s concolic execution engine is invoked a second time, generating enough new inputs to pass these checks. From this point, the fuzzer is able to find 271 additional basic blocks within a large compartment of the application that processes generic input which, for this application, consists of parsing code

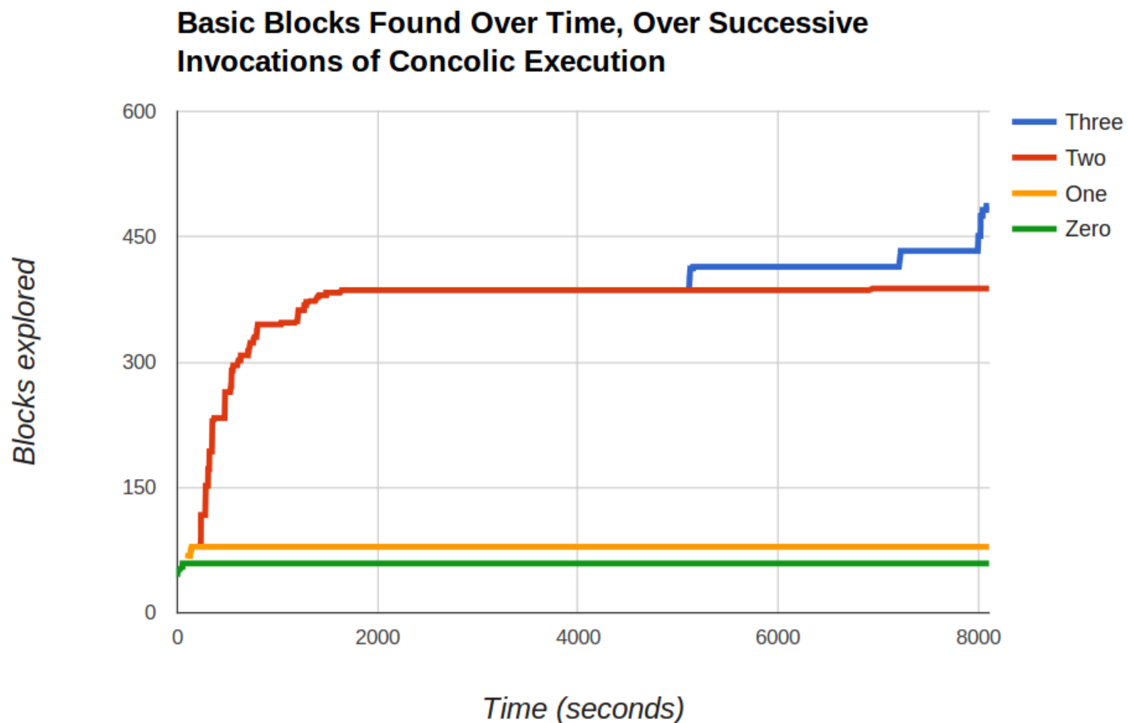


Figure 3.9: For the binary 2b03cf01, which Driller crashed in about 2.25 hours, this graph shows the number of basic blocks found over time. Each line represents a different number of invocations of symbolic execution from zero to three invocations. After each invocation of symbolic execution, the fuzzer is able to find more basic blocks.

relating to analysis of the user-provided electrical design. Eventually, the fuzzer finds all of the interesting paths that it can in that compartment and decides that it is not making further progress, leading to another invocation of Driller's concolic execution engine.

This time, Driller finds 74 new basic blocks and generates inputs that reach them by successfully passing checks on the input that the fuzzer had not previously satisfied. These additional basic blocks (represented by the black nodes in Figure 3.10) comprise the functionality of adding specific circuit components. For the interested reader, Listing 3.9 presents one of the functions that contains specific checks against user input with which fuzzers have trouble. Input representing these components must adhere to an exact specification of a circuit component, and the checks of these specifications is what the third invocation of Driller's concolic

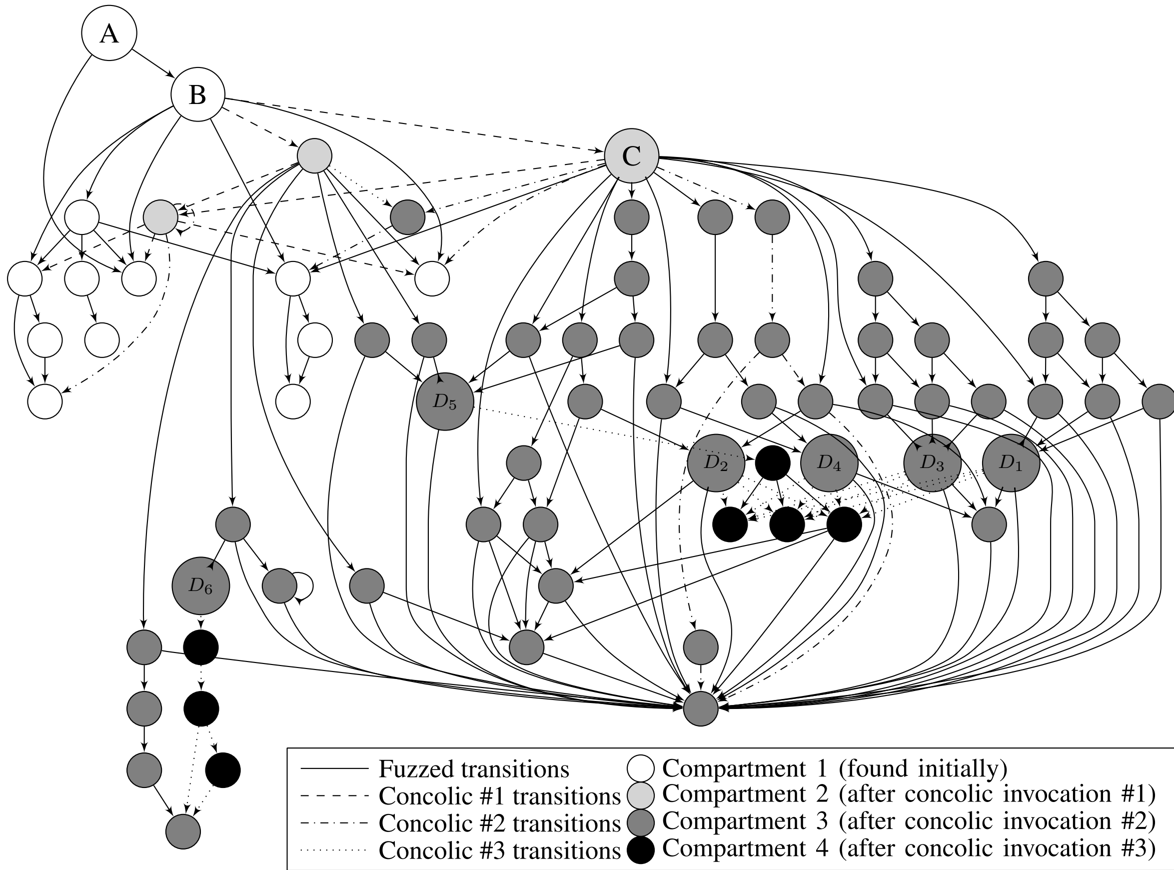


Figure 3.10: Graph visualizing the progress made by Driller in discovering new compartments. Each node is a function; each edge is a function call, but return edges are excluded to maintain legibility. Node “A” is the entry point. Node “B” contains a magic number check that requires the symbolic execution component to resolve. Node “C” contains another magic number check.

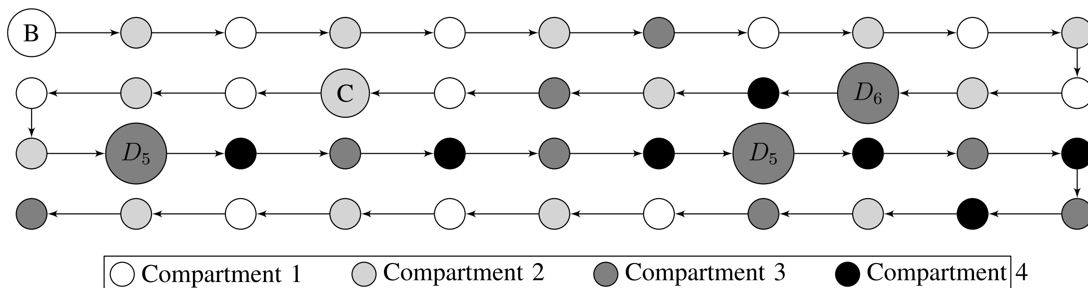


Figure 3.11: The sequence of compartments through which execution flows for a trace of the crashing input for CGC application 2b03cf01. Driller’s ability to “break into” the fourth compartment (represented by the black nodes) was critical for generating the crashing input.

Listing 3.7: The first complex check in the 2b03cf01 application.

```
1 enum {
2     MODE_BUILD = 13980,
3     MODE_EXAMINE = 809110,
4 };
5
6 ...
7
8 RECV(mode, sizeof(uint32_t));
9
10 switch (mode[0]) {
11     case MODE_BUILD:
12         ret = do_build();
13         break;
14     case MODE_EXAMINE:
15         ret = do_examine();
16         break;
17     default:
18         ret = ERR_INVALID_MODE;
19 }
```

execution engine finds. These constants that are used in this function's checks are defined in the code reproduced in Listing 3.8. A fuzzer cannot guess these constants without exhausting a huge search space, as they are specific values of 32-bit integers. Driller's symbolic execution, however, can find these constants easily, since the comparisons in the code produce easy-to-solve-for conditions on paths taking these branches.

Listing 3.8: An enum definition with explicit constants. In order to guess these constants, these specific values must be guessed from a search space of 2^{32} numbers.

```
1 typedef enum {
2         FIFTEEN_AMP = 0x0000000f,
3         TWENTY_AMP = 0x00000014,
4 } CIRCUIT_MODELS_T;
```

Listing 3.9: A function with a switch statement testing user input against a number of specific values

```
1 int8_t get_new_breaker_by_model_id(CIRCUIT_MODEL model_id, breaker_t *breaker_space,
2                 uint8_t breaker_space_idx) {
3         int8_t res = SUCCESS;
4         switch(model_id) {
5                 case FIFTEEN_AMP:
6                         create_breaker(15, breaker_space, breaker_space_idx);
7                         break;
8                 case TWENTY_AMP:
9                         create_breaker(20, breaker_space, breaker_space_idx);
10                        break;
11                default:
12                        // invalid model_id
13                        res = -1;
14        }
15        return res;
16 }
```

Driller's new input is then passed back to the fuzzer in order to quickly assess the new coverage generated by the change. Listing 3.10 shows some of the code that executes for the first time as a result of the new input. The user input that this new component processes is no longer specific, but general, making it suitable for the fuzzer. From this point on, the fuzzer continues to mutate these inputs until it triggers a vulnerability caused by a missing sanitization check in the application.

Listing 3.10: Code executed as a result of passing the specific check

```
1 static void create_breaker(uint8_t amp_rating, breaker_t *breaker_space,
2                 uint8_t breaker_space_idx) {
3         breaker_space->id = breaker_space_idx;
4         breaker_space->amp_rating = amp_rating;
5         breaker_space->outlets = list_create_dup();
6         if (breaker_space->outlets == NULL) {_terminate(ERRNO_ALLOC);}
7 }
```

In a semantic sense, the vulnerability involves initializing a new breaker object in the cir-

cuit the user creates. Later on, the circuit will be tested for connectivity, among other things, and component-specific logic will be invoked depending on the materials of which the circuit is composed. Satisfying the check to add a breaker will expand the bug-searching coverage to include breaker-specific code. Triggering this vulnerability requires the inclusion, in the provided circuit diagram, of specifically crafted breaker components. The inputs required to trigger the creation of these components are what Driller recovers in the third concolic execution invocation, and the final fuzzing invocation mutates them enough to trigger the vulnerable edge case.

The final path taken by the crashing input is shown in Figure 3.11. Starting at the entry point, this path goes through progressively harder-to-reach compartments (represented by the different colors of the nodes) until the condition to trigger the edge is created. This binary was not crashed in either baseline experiment – the unaided fuzzer was never able to reach compartments of the code "protected" by the complex checks, and the symbolic exploration engine experienced an almost immediate path explosion in the input-processing code. By combining the merits of fuzzing and concolic execution, Driller was able to crash this binary in approximately two and a quarter hours.

We present the amplification in basic block coverage that each concolic execution invocation produces in this binary, plotted over time, in Figure 3.9.

3.6 Discussion

Driller carries out a *unified* analysis by leveraging both symbolic execution and fuzzing. This allows Driller to address some of the drawbacks of each analysis with its complement. In this section, we will discuss the limitations of Driller and future directions of research to further augment automated vulnerability extraction.

3.6.1 Limitations

Both a benefit and pitfall of Driller is its borrowing of state-space interpretation from AFL. AFL represents state simply by tracking state-transition tuples to rough “hit counts” (how many times the state-transition was encountered). This moderately light representation of state is what allows AFL to be so efficient as each path’s state is only defined by the collection of state-transition tuples it encountered combined with how many times they were encountered. Driller uses this same data structure to determine which state transitions are worth solving for. We provide an example of how this can limit Driller in Listing 3.11

Listing 3.11: An example of minimal state representation limiting discovery of new state transitions.

```
1  int static_strcmp(char *a, char *b){
2      for (;*a;a++,b++) {
3          if (*a != *b)
4              break;
5      }
6
7      return *a - *b;
8  }
9
10 int main(void) {
11         read(0, user_command, 10);
12
13         if (static_strcmp("first_cmd", user_command) == 0) {
14         cmd1();
15         }
16     else if (static_strcmp("second_cmd", user_command) == 0) {
17         cmd2();
18     }
19     else if (static_strcmp("crash_cmd", user_command) == 0) {
20         abort();
21     }
22
23         return 0;
24 }
```

This listing demonstrates a state-transition which occurs in multiple command handlers. Since each branch relies on `static_strcmp`, AFL itself will not be able to distinguish between state-transitions inside different invocations of `static_strcmp`. Driller uses the same metric to determine which state-transitions need to be solved. As such, Driller will not try to solve for the `if` statement on line 3 more than once, even though it is used for different comparisons. Additionally, inputs which have one or two additional matching characters would not be considered interesting by AFL. Of course if the entire string was discovered by Driller,

AFL would find it interesting and adopt it. Driller attempts to mitigate the effects of this problem with the symbolic explorer stub (described in 3.4.4) invoked at each new state transition. However, we believe this is an imperfect solution and ultimately a better representation of state might be required.

Another limitation of Driller is the case when user input is treated as *generic* input in one component and *specific* input in another. Consider the program presented in Listing 3.12.

This application reads a command and a hash from the user and verifies the hash. This compartment, spanning lines 1 through 11, treats the command as generic input and the hash as specific input. After this, however, the application checks, in multiple stages, that the provided command was "`CRASH!!`". Fundamentally, this reclassifies the `user_command` as specific input, as it must be matched exactly. This triggers a case that reduces Driller to a symbolic execution engine, as explained below.

Listing 3.12: An example of input being used as generic input in one place and specific input in another. A crashing input for this binary is "CRASH!!" followed by its hash.

```
1 int main(void) {
2         char user_command[10];
3         int user_hash;
4 
5         read(0, user_command, 10);
6         read(0, user_hash, sizeof(int));
7 
8         if (user_hash != hash(user_command)) {
9                 puts("Hash mismatch!");
10                return 1;
11        }
12
13        if (strncmp("CRASH", user_command, 5) == 0) {
14                puts("Welcome to compartment 3!");
15                if (user_command[5] == '!') {
16                        path_explosion_function();
17                        if (user_command[6] == '!') {
18                                puts("CRASHING");
19                                abort();
20                        }
21                }
22        }
23
24        return 0;
25 }
```

Passing through the first stage, into compartment 3, is straightforward – Driller's concolic execution engine will identify an input that starts with "`CRASH`" and its corresponding hash (as this is a forward-calculation of a hash, there is no concern with having to "crack" the hash; Driller merely needs to calculate it). However, after this, the fuzzer will no longer function for exploring this compartment. This is because any random mutations to either the hash or the input will likely cause execution to fail to proceed from compartment 1. Thus, the

fuzzer will quickly get stuck, and Driller will invoke the concolic execution engine again. This invocation will guide Driller to compartment 4, on line 16, and hand execution back to the fuzzer. However, the fuzzer will again fail to proceed, decide that it is stuck, and trigger concolic execution.

This cycle will continue, making the fuzzing component useless and essentially reducing Driller to symbolic exploration. Worse, in this application, compartment 4 calls a function (`path_explosion_function`) that causes a path explosion. Without the mitigating effects of its fuzzing engine, Driller is unable to reach compartment 5 (lines 18 and 19) and trigger the bug.

This represents a limitation in Driller: in certain cases, the fuzzing component can become effectively disabled, robbing Driller of its advantage. A potential future step in mitigating this issue is the ability to generate "semi-symbolic" fuzzing input. For example, the concolic engine might pass a set of constraints to the fuzzer to ensure that the inputs it generated conform to some specification. This would take advantage of the concept of *generational fuzzing* [54] to create "input generators" to aid the fuzzer in reaching and exploring application compartments.

The limitation exemplified by Listing 3.12 shows how a specific input can prevent the fuzzer from effectively mutating the generic input. However, for other types of specific input, even with multiple components, AFL can still fuzz the deeper components. Even in the most difficult cases, such as hash checks, Driller will still be able to mutate any input that is unrelated to the hash, such as input after the hash is checked. We do expect some decrease in performance after Driller has found multiple components. This is because AFL has no knowledge of the constraints from the symbolic execution engine, so there will be a fraction of the fuzzing cycles wasted trying to mutate specific inputs.

3.7 Conclusion

In this paper, we presented Driller, a tool that combines the best of dynamic fuzzing and concolic execution to efficiently find bugs buried in a binary. We introduce the concept of a *compartment* of a binary, which largely separate functionality and code. Within Driller, fuzzing provides a fast and cheap overview of a compartment, effectively exploring loops and simple checks, but often fails to transition *between* compartments. Selective concolic execution gets into state explosions when considering loops and inner checks, but is highly effective at finding paths between compartments of a binary. By combining these two techniques, where each individually fails, Driller is able to explore a greater space of functionality within the binary.

We evaluated Driller on 126 binaries from the DARPA Cyber Grand Challenge Qualifying Event. Driller found 77 crashes, a substantial improvement over basic fuzzing's 68 crashes. We believe the technique shows promise for general-purpose bug-finding in all categories of binaries.

Acknowledgments

We would like to thank all contributors to the DARPA Cyber Grand Challenge organization (for providing an excellent testing ground for our tool), Michal Zalewski (his public tool, AFL, and documentation proved immensely useful), Secure Business Austria, the contributors of angr, and of course, all our fellow Shellphish CGC team members, *donfos* in particular. This material is based on research sponsored by DARPA under agreement number N66001-13-2-4039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S.

Government.

Chapter 4

Uncooperative Remote Runtime Patching

4.1 Introduction

The modern world is run by interconnected software. Software handles our communications, manages our finances, and stores our personal information. In addition, with the rise of the Internet of Things (IoT), the number of embedded devices running complex software has skyrocketed [55]. In fact, the number of bugs found in software has been increasing over time [35]. Leveraging these bugs lets an attacker perform actions ranging from the theft of money or data to, in the case of the Internet of Things, influence the physical world.

The common approach to remedying buggy software is *patching*. However, patches suffer from very slow *adoption* by users, in part because many patches require system restarts to be applied or to take effect [56, 57]. In the IoT world, the situation is even more problematic, as device vendors often fail to incorporate effective and easy-to-use means to update their products. As a result, even when a vulnerability is found and publicly disclosed, it is difficult (or even impossible) for users to install these patches. Finally, the IoT market is a volatile space, with vendors entering and leaving the ecosystem. This means that a vendor might not be around anymore while its vulnerable devices are still connected to the network.

In the best case, when a vulnerability is discovered, the responsible software vendor will simply develop a patch and push it to its users to secure their devices. Unfortunately, this scenario does not always play out. As mentioned above, a device might lack update functionality, users might not understand how to apply patches (for example, when firmware must be flashed), or the software vendor is no longer present. In all these cases, we would like a mechanism that is able to “force” a patch onto the vulnerable system and fix the vulnerability.

In this paper, we present a technique, called *Piston*, that leverages the presence of bugs to automatically *patch* a system as the result of exploiting these vulnerabilities. By leveraging an exploit to patch software, Piston has the unique ability to patch applications without direct privileged access or, in fact, *without any access to the host at all*. Of course, exploiting a vulnerability in a target process and using this access to patch the underlying vulnerability raises a number of questions and poses significant challenges:

First, not all bugs can be used for patching – it must be possible to take control of the victim process. “Fortunately,” a significant portion of bugs manifest as *memory corruption* leading to *control-flow hijacking* [36]. Our intuition is that, aside from taking control of a process for nefarious purposes, a control-flow hijack can be leveraged to achieve remote hot-patching of buggy software.

Second, leveraging an exploit to forcefully take control over a process can have adverse effects on the execution of this process, such as causing a crash. In some cases, this would not be a problem. That is, Piston could take control over the process, update the vulnerable application on the system (on persistent storage), and restart it. Unfortunately, this approach does not always work. One problem is that the running process might not have the privileges to write to the permanent storage, and hence, cannot make the patch persistent. Another problem is that the software might control a critical process, and interrupting its execution has unintended and unwanted consequences. Hence, it is critical that we perform the patch in a way that allows the process to continue its execution without interruption (longer than it takes to patch the running

code) or even a crash.

Previous work has introduced the idea of *hot-patching*; a system able to apply patches to software while the software is running. Such systems have been developed for vehicles [58], kernels [59], user-space software [60], and, in general *uninterruptible systems*, or systems where correctness depends on their continuous execution. However, these approaches typically have two requirements: fore-planning on the part of the author, and privileged access to the computer running the software. For example, kernel hot-patching systems, such as KSplice [59], require a custom kernel module to be loaded, which requires administrative privileges. Unfortunately, a large amount of software does not meet these requirements. User-space software rarely supports updates without a restart, and many embedded devices do not give the user necessary permissions. To remedy such situations, a new approach to patching is required. Specifically, Piston uses novel applications of binary analysis to identify and automatically repair data that is corrupted as part of the exploit. That is, our system exploits a vulnerability to take control of the running process, repairs the damage that this exploit has caused, then patches the bug in the code, and finally lets the (now secure) process continue to execute.

Here we will talk about four distinct applications of Piston.

Patching uncooperative systems. Certain systems, such as embedded devices, require that updates be created and distributed by the device manufacturer. This poses a problem to end users: these patches are often only provided for a limited period of time, are produced very slowly, or are never produced at all (in fact, many smaller embedded devices lack any sort of update mechanism). Even when patches are distributed, it might be inconvenient to apply them. Some devices need to be physically connected to a computer to apply the update, and reboots are standard in almost all cases. Piston allows these devices to be updated remotely, as long as the original firmware has a vulnerability that can lead to code execution. The systems do not have to be designed to be hot-patched with Piston, unlike with prior approaches for

hot-patching embedded systems.

Patching continuity-critical systems. In some applications, downtime can be prohibitively costly, or even mean the difference between life and death. A couple examples of these applications are critical infrastructure components and medical devices. If a vulnerability is discovered in such systems, it may take significant time before an update can be applied in a safe, scheduled maintenance window. As such, systems that have not been developed with hot-patching in mind, may remain vulnerable to exploits for quite some time as the maintainer prepares for the downtime to apply the patch.

Piston can instead use this vulnerability (if it leads to code execution) to provide the update while the system is running. This can reduce the potentially dangerous delay, as well as preventing the need to schedule emergency maintenance downtime in such cases.

Emergency patching. As more of our personal and business dealing moves online, security becomes paramount. Whereas compromises may have been simply embarrassing to an organization a decade ago, today they can cause serious damage to companies. Thus, organizations must patch software flaws as soon as possible. However, many organizations struggle to roll out security updates. If properly used, Piston could make them easier. Piston could be used as a first-stage emergency patching system. In our example detailing a patch of `NGINX` in Section 4.6, a company running internet-facing `NGINX` services could scan their entire network and use Piston to apply ephemeral (in-memory) emergency patches to every vulnerable host to tide them over until a permanent patch can be deployed.

Helpful worms. Users and businesses are slow to update devices, often leaving machines vulnerable long after patches are available. For example, the Wannacry ransomware exploited a flaw for which a patch was available three months earlier [61]. Previous work has explored using "helpful" worms to apply patches on a large scale, without users or admins needing to apply the patch [62]. One example is the *Welchia worm*, known for removing the harmful Blaster worm and patching the device. Piston could enable the creation of these helpful worms,

even when the vulnerable process does not have enough privileges to apply the official patch, by applying the patch in-memory.

In this paper, we describe Piston's approach and detail its implementation atop an existing open-source binary analysis framework [37]. We discuss situations in which Piston can operate automatically and semi-automatically, and evaluate its efficacy on a handful of binaries from DARPA's Cyber Grand Challenge with documented vulnerabilities. We target stack overflows in these binaries and show Piston's effectiveness at automatically remotely patching through a memory corruption exploit. Additionally, we demonstrate Piston's applicability by remotely patching NGINX 1.4.0 against CVE-2013-2028 [63], using that same vulnerability to achieve remote code execution. We do this to show that Piston can be used on complex, real-world binaries with very little analyst intervention.

In summary, this paper makes the following contributions.

Remote hot-patching. We detail our design for an automatic, remote hot-patching system, called Piston, which generates patches from compiled binaries.

Recovery from an exploit. We introduce novel techniques to automatically recover a program's state and continue execution after an exploit.

Evaluation. We use a set of binaries from the DARPA Cyber Grand Challenge to evaluate Piston's effectiveness at achieving automated remote patching through the exploitation in addition to evaluating Piston's application to real-world, commonly deployed software, such as NGINX.

4.2 Overview

Piston is not the first approach to patching computer software at runtime, a process known as *hot-patching*. In this section, we will give a general overview of Piston, and its novelties,

before moving on to describe the individual steps in detail in the next section.

Unlike previous work, Piston is designed to patch uncooperative systems *remotely*. As the systems it targets are not designed to be patched in this way (hence *uncooperative*), this patching requires a level of remote access unintended by the authors of the software being patched. Piston achieves this access through the use of an *exploit*. This adds two significant challenges to the patching process. First, unlike existing hot-patching systems, patching must be performed during the exploitation of the vulnerable process, rather than selecting easy patch points. Second, the exploitation of the target process frequently damages that process' memory space. To allow the program to continue executing, Piston must *repair* the memory space of the program after the patch is applied, all while the software is running.

Unlike some prior work, Piston functions directly on binaries, with no access to source code. This allows Piston to work on proprietary software without source code from the vendor, but also makes its work more complicated, as a substantial amount of relevant information is lost when a binary is compiled.

Piston has four pre-requisites for its operation:

Original binary. This is the binary program that is currently running as the remote process or system.

Replacement binary. This is the "patched" binary. The remote process will be functionally updated to this version of the binary after Piston's operation.

Exploit specification. Piston expects a description of how to trigger a vulnerability in the remote process. This specification must be able to achieve code execution in the remote process, which Piston will use to apply the patch. The exploit is expected to bypass common mitigations such as ASLR and NX if they are used on the target system.

Remote configuration. To properly model the environment of the remote process, Piston

needs to have a specification of its configuration. For example, if the remote process is an `nginx` web server, its configuration file must be provided.

Given these inputs, the approach has three major steps:

1. Patch generation. Given its inputs, Piston performs in-depth static analysis of the binary to identify the "patch" that needs to be applied in the memory of the remote process. This is done by leveraging binary diffing techniques, which is discussed in detail, in Section 4.3.

2. Repair planning. Unlike traditional hot-patching systems, Piston *exploits* a process in order to patch it. Thus, Piston faces a unique challenge: in the course of exploiting the remote process, the memory state of the remote process might be damaged.

Piston has the capability to automatically generate a routine which repairs the corrupted state of a process if it was exploited with a stack-based buffer overflow. For cases which Piston cannot repair automatically, including other types of exploits, Piston will require the analyst to provide a *repair routine* that should repair the parts of the process' memory that Piston is unable to restore. Piston, can report the parts of the state that were corrupted to the analyst to aid in the creation of the repair routine. In our evaluation (see section 4.6) we show that this repair routine can be automatically generated in the majority of stack-based buffer overflows which we tested.

Piston may also require a *rollback* routine that undoes the partial effects of functions that were interrupted by the exploit. In the case where a patch involves making a change to a structure definition, Piston requires an analyst to supply a *state transition routine*. This routine should be responsible for updating all instances of the structure in the target's memory to abide by the newly patched-in definition.

We talk in-depth about cases where Piston can fully automatically repair the state and cases where analyst intervention is necessary in Section 4.4.

3. Remote patching. Piston uses the exploit specification to craft an exploit to inject the

patcher core. The patcher core, running in the remote process, retrieves the patch information, a state transition routine, a rollback routine, and a repair routine. Piston may deem any one of these routines to be unnecessary to the hot-patching process, with the exception of the state transition routine where an analyst is responsible for judging its necessity.

Piston uses the patcher core to then apply these received routines in turn. After this is completed, the execution returns to the now-patched remote process and Piston's operation is complete. In-depth details of the patching step are in Section 4.5. When it terminates, the remote process will be running a codebase that is functionally equivalent to the patched binary.

4.3 Patch Generation

Piston receives, as input, the original binary representing the target process and the replacement binary to which the target process should be updated. Given these binaries, it must identify specific patches that must be applied in order to accomplish this update.

Similar to other systems, such as Ksplice, Piston applies patches on a function level rather than replacing the entire binary in the remote process. If a function is updated in the replacement binary, its counterpart in the remote process (running the original binary) will be replaced at runtime. If a function is found to be *unique* to the replacement binary, it will be added to the remote process.

Additionally, in the updated binary, addresses of code and data will usually change. Therefore any references to code and globals must be updated in the replacement functions. Piston will fix the references to point to those in the currently running process.

Piston's preprocessing works in several stages:

1. Piston matches updated functions between the original and replacement binaries. The matches are filtered to eliminate superficial differences.

```
push    ebp                              push    ebp
mov     ebp, esp                         mov     ebp, esp
sub     esp, 0x18                        sub     esp, 0x18
mov     eax, 0x804a02c                   mov     eax, 0x805a084
:                                        :
0x804a02c  "Hello %s"                    0x805a084  "Hello %s"
```

Figure 4.1: Superficial difference example

2. Piston chooses a location in the memory space of the remote process, in which, to place remaining updated functions. These functions are "fixed up" to allow them to function in the memory space of the remote process and run in the context of the original binary.

The output is a *patch set* (represented as a diff of memory) that Piston will apply to the remote process in the remote patching step.

4.3.1 Function Matching

First, Piston must identify the functions that need to be updated or added to the remote process. This requires Piston to understand which functions in the original binary correspond to which functions in the replacement binary. We leverage existing binary diffing techniques [64] for this step, allowing us to support the correlation of functions even when there are no symbols in the binaries. These techniques work on the control flow graph, so they are robust to small compiler artifacts.

At the end of this stage, Piston generates a set of pairs of matching functions and a set of introduced functions. Additionally, a candidate set of original function to replacement pairs is constructed by checking for differences in the content of matched functions.

Piston's initial candidate set of updated functions contains some false positives. This is because any change in the length of code will cause addresses to be different in the replacement binary; in turn, the differing addresses will show up as changes in the operand of instructions. We consider all references to the same code or data between a pair of matched functions to be superficial. An example of a superficial difference is shown in Figure 4.1.

Thus, Piston *filters* this set of updated functions to remove superficial changes. If an updated function contains only superficial changes, we discard it and its match from the candidate set. The remaining members of the candidate set, along with the introduced functions, are the ones that Piston will patch into the new binary.

4.3.2 Replacement Function Placement

Because Piston replaces individual functions rather than the entire binary, it runs into the challenge of function placement. As previously discussed, replacement functions may be larger than their original counterparts. Because of this, Piston chooses a new address in the executable memory space of the remote process to place replacement functions. This requires an addition function fix-up step: any relative references in the function will need to be updated to compensate for the new location.

New functions or data that were not in the original binary can be resolved by adding the code or data to the new process. Newly added code and data might also have references to other code and data, so it needs to be handled similarly until all references are resolved.

After determining a place for the replacement function in this new area of memory, Piston places a trampoline (direct jump instruction) at the beginning of the old function. Piston checks that the trampoline will fit entirely inside of the first basic block of a function to ensure that execution will never jump to the middle of an instruction. This is useful for several reasons. First, it lets us replace the function while keeping all references to it, such as function pointers and direct calls from other places in the code. Second, if there are any return addresses to code inside the original function on the stack, they remain valid, although the patched code will not be executed until the function returns. Note that this means that Piston can only patch functions that will eventually return, and infinite looping functions, such as a main loop, cannot be patched. This minor limitation is also common among other hot-patching systems.

4.4 Repair Planning

Piston achieves the hot-patching of a remote process by leveraging an exploit to achieve code execution in the context of the process, and then using this capability to inject patched code before resuming process execution. Unfortunately, exploits typically cause the *corruption* of the memory space of the remote process, and resuming a process after such corruption can be non-trivial.

For example, during the exploitation of a stack-based buffer overflow, process data on the stack is overwritten with either shellcode or a ROP chain, ultimately leading to the hijacking of control flow by the exploiter. If this memory corruption is not corrected before execution is resumed, the process will simply crash. To remedy this, a *repair* step is required before Piston can resume the patched process.

Piston can fully automatically generate a repair routine in cases of stack-based buffer overflows. This automatic approach uses redundant data in memory and registers to restore the state of the process. In principle, this approach applies to any corruption, not just that on the stack. However, empirically, we have not found an adequate level of data redundancy in other classes of exploits, and therefore, require the analyst to provide the repair routine if the exploit is not a stack-based buffer overflow. We discuss the redundancies inherent in stack data in Section 4.4.3 and the limitations in repairing other corruption in Section 4.7. We focus on buffer overflows in Piston's current implementation, as they still represent the third most common type of vulnerability in *all* software [36]. Furthermore, a recent analysis of trends in CVE's found that buffer overflows rank the highest for severity, and that buffer overflows are the second most common vulnerability that applies to binary software, behind denial of service vulnerabilities [65].

Piston carries out an offline analysis of the original binary and the exploit specification to assess the damage that an exploit causes and creates such a repair plan. This analysis is

done *off-line*, before the patching process itself, and the repair plan is applied to the remote process after it is patched. The on-line patch application step is discussed in Section 4.5. Piston assumes that the exploit it will use to patch the remote process will hijack execution either partway through a function or at the return point of a function. We term this the *hijacked function*, and reason about exploitation after-effects as they relate to this function. We name the function that calls the hijacked function the *caller function*.

Piston can restart the remote process after patching if the following conditions are met:

1. The hijacked function either completes successfully (i.e., the exploit does not influence its operation and simply hijacks the control flow when it returns), or its effects (such as memory writes) can be analyzed and *undone* and the function can be restarted. In the former case, Piston can simply return to the caller function after the remote process is patched. However, in the latter case, the effects of the hijacked function, such as the modification of memory and registers, must be undone. After undoing this modification, Piston can return to the *call-site* of the hijacked function, and trigger its re-execution after the patching is complete.

2. Any state of the caller function that was corrupted (such as local variables within in the stack frame) can either be recovered or is not needed after the patched process resumes. If the caller function has corrupted state that cannot be recovered, Piston can try to treat the caller function as *hijacked* and, instead, undo its effects and try to restart it. In this case, Piston's recovery process is repeated with the prior caller function being the new hijacked function and the *caller* of the original caller function to be the new caller function.

To meet these requirements, Piston creates a repair plan that includes two routines that will be executed inside the remote process after it is exploited. These routines are the *rollback routine*, which will undo the actions of the hijacked function (if necessary), and the *repair routine*, which will restore the local state of the caller function to be non-corrupted.

Automatically generating these routines represents a significant challenge, and there are two cases when manual analyst intervention might be required. First, depending on the complexity of the hijacked function, Piston might be unable to automatically undo its effects. In this case, the analyst must manually provide the *rollback routine* that will be run in the remote process before the hijacked function is restarted.

Second, the exploit might cause irreparable damage to the caller function's state. In this case, Piston provides two options to the analyst: the analyst can manually provide a *repair routine*, or Piston can attempt to undo and restart the caller function as well. To do so, it moves further up the callstack, classifying the caller function as the new hijacked function and that function's caller as the new caller function and repeating its analysis.

Piston creates the repair plan in three steps:

Exploit effect reconstruction. To reason about the state of the remote process after exploitation, Piston carries out the exploit against the original binary in an instrumented environment. The trace that is created during this step is used in further analyses.

Hijacked function analysis. Piston analyzes the exploit trace to determine whether the hijacked function had successfully completed its work. If the hijacked function was interrupted, Piston must annul the function's effects and restart it after the patch completes. To understand how to properly undo the effects of the function, Piston performs an in-depth analysis of the function using symbolic execution techniques.

Caller state recovery. Next, Piston determines the extent of state clobbering outside of the hijacked function's stack frame by analyzing the exploit trace. It attempts to create a *state repair plan* for this damage, leveraging symbolic execution of the caller function to identify uncorrupted parts of the state that can be used to restore corrupted values.

Different types of exploits cause different damage to the remote process. For example, a simple pointer overwrite might not require much memory repairing, whereas a stack overflow

can corrupt much of the stack. In its current state, Piston can automatically create a repair plan for memory corruption resulting from most stack-based buffer overflows. Piston automatically detects corruption resulting from stack-based overflows as well as heap-based overflows, but automatically supporting corruption detection for additional exploits simply requires a routine to recognize the corruption they cause (i.e., expanding the processes described in Sections 4.4.1 and 4.4.1).

4.4.1 Exploit Effect Reconstruction

Piston generates an *exploitation trace* to reason about the damage that the exploit will cause to the remote process. The exploitation trace is created by executing the original binary (configured with the remote configuration), using the exploit specification as input. During the trace, control flow transitions and writes to and reads from registers and memory are recorded for future analysis.

Detecting the Exploitation Point

To understand what repairs are needed after exploitation, Piston must classify memory writes based on whether or not they are a result of the exploit or of the intended operation of the binary. Piston does this by identifying the *exploitation point*. Intuitively, the exploitation point is a point in the trace after which the process can no longer be considered to be operating properly.

For stack-based buffer overflows, Piston uses a simple heuristic to identify this point: Piston tracks all saved return addresses and callee-saved registers throughout execution. When one of these is overwritten, Piston assumes that it has identified the exploitation point. The function where this exploitation point takes place, is the *hijacked function*.

For heap-based overflows, Piston tracks calls to heap allocation and deallocation functions

such as `malloc()`, `realloc()` and `free()`. [1] During the *exploitation trace*, Piston keeps a list of the heap buffers and updates it at every call to these functions. At every write to the heap, Piston checks whether or not the address resides in one of these buffers. If the address does not reside in any such buffer, it is assumed that the exploitation point has been identified.

One caveat of these heuristics is that Piston cannot identify the exact exploitation point for exploits which perform the overflow entirely within a stack frame, or within a struct on the heap. Although advanced type analysis can automatically infer the data types and structure of objects and stack frames [66] [67], such analysis is out of the scope of this paper. Piston can be extended with additional routines in order to support automatic detection of corruption of other exploits such as these.

Identifying Corruption

Once the exploitation point has been identified, Piston can determine the parts of the program state that were corrupted. Again, a heuristic specific to buffer overflows is leveraged: Piston marks as "corrupted" all data that was written to the buffer that was overflowed. This step is done *retroactively* by analyzing all of the writes to memory that occurred.

Piston uses a simple heuristic to identify buffers: it assumes that all writes that are initiated by the same instruction (not the same invocation of that instruction, but all invocations) are writes to the same buffer. This approach is inspired by the buffer detection proposed by MovieStealer [68], which groups buffers by *loops* instead of instructions. Piston marks all writes by the same instruction during the invocation of the hijacked function as writes to the same corrupted buffer. We term this instruction the *overflow instruction*.

[1] In statically linked binaries, such as firmware, an extra step is necessary identify these functions as they may not contain symbols. We use test cases, comprising of input states and expected outputs to identify these functions as described in [37].

Exploitation Trace Soundness

It is possible that the analysis results might not perfectly match the state of the remote process during exploitation. For this reason, Piston augments the trace with more general analyses in other steps and only assumes that two pieces of information from the trace are accurate:

1. The exploit will overflow the buffer by the same number of bytes in the exploitation trace as it will when run against the remote process.
2. The hijacked function and its caller function will be the same on the remote process as in the exploitation trace.

We have not seen a case that violates either of these assumptions, but it is a theoretical possibility.

4.4.2 Hijacked Function Analysis

Having identified the hijacked function and the range of the corrupted data, Piston must next determine whether the hijacked function terminated successfully or whether it needs to be restarted.

Conceptually, the determination of whether the hijacked function terminated successfully is simple: Piston considers the function as having successfully completed if it can show that no action was taken based on corrupted data. This happens fairly frequently: modern compilers tend to avoid placing local variables after buffers in memory, since doing so would allow the local variables (instead of just the return address) to be overwritten by a buffer overflow, potentially allowing the attacker to influence program behavior even before the function returns. However, in cases where this is not the case (either because the compiler did not choose such a placement or because there is more than one buffer on the stack), we consider the hijacked

function's operation to have been *interrupted*, and Piston must undo the corrupted effects and restart the function after patching.

Checking for Successful Completion

Functions have memory that can be written to without influencing the operation of the remainder of the program; we call this memory *scratch space*. Scratch space is considered to be the local stack frame as well as any memory regions which are freed before the return site of the function. Data in these ranges will not be used outside the function in a well-formed program. Other data, such as globals, heap data which is not freed inside of the function, and return values may influence the remainder of the program and are not considered scratch space.

Piston determines the successful completion of the hijacked function during the dynamic analysis of the offline exploitation trace. At the exploitation point, the corrupted data range is marked as tainted, and the taint is tracked through the remainder of the function. If any branch is influenced by tainted data or if tainted data is written outside of the scratch space, then the function is considered to have not completed successfully. We filter out the restoration of callee-saved registers at the end of the function, as these are considered part of the state of the caller and will be restored later.

Checking for Repeatability

When Piston is unable to prove that the hijacked function completed successfully, it will check if its execution can simply be repeated after the remote process is patched. The hijacked function can be safely restarted if all of the inputs (i.e., values the function reads from memory or registers) to the interrupted invocation can be recovered. If these inputs can be recovered, the hijacked function can be re-executed in the same context as its interrupted invocation and will carry out the same actions.

Piston groups the inputs that a function receives into three categories: *local state data*,

which is passed to the hijacked function on the stack or in registers, *global state data*, which is retrieved by the hijacked function from the heap or global memory, and *environment inputs*, which are retrieved through system calls. The hijacked function is considered *repeatable* if these conditions hold:

Local state data is recoverable. Arguments on the stack, which might be clobbered during the overwrite itself or by actions taken by the hijacked function after exploitation, must be recoverable, as must arguments passed to the function through registers. This recovery is explained in Section 4.4.3.

Global state data is recoverable. All data in registers and memory that the hijacked function reads must be recoverable. This means that the hijacked function cannot irrecoverably overwrite its inputs.

System calls are repeatable. The system calls invoked out by the hijacked function must be repeatable. System calls that cannot simply be re-executed, such as `unlink` (since, after the first call, the file will no longer exist), violate this condition.

The first condition will be checked during the caller state recovery step. To check the latter two conditions, Piston collects a list of all memory accesses and system calls in the exploitation trace, which can then be checked for any violations to the repeatability conditions.

To detect changes to the global state, the list is analyzed to build a set of any potentially corrupted global state by the original run of the hijacked function. After this, the list is analyzed again to see if any of the corrupted global state can be used as an input to the repeated invocation of the hijacked function. Conceptually, this happens when the hijacked function reads in some global value before writing to it (for example, incrementing a global counter). The second invocation will use the value corrupted by the first, resulting in an inconsistency in execution between the interrupted invocation and repeated invocation of the hijacked function.

Piston will attempt to undo simple global changes where no dereference of data takes place. We use under-constrained symbolic execution (UCSE), an extension of dynamic symbolic execution that enables the analysis of functions without the requirement of *context* [69]. UCSE works by identifying memory dereferences of pointers that are unknown due to missing context (for example, a pointer that would have been passed as an argument) and performs on-demand memory initialization to allow the analysis to continue.

Piston explores the hijacked function with UCSE, *ignoring* the context from the exploitation trace to avoid under-approximating the remote state. If UCSE can determine how a global value will change during execution of the hijacked function, then Piston can recover the value automatically. Note that, like other techniques based on symbolic execution, UCSE can succumb to path explosion. When this occurs, Piston will be unable to automatically recover changes to global state. If any changes to global state are detected that Piston is unable to recover, analyst will be required to provide a *rollback routine* to undo the effects of the interrupted execution.

After checking for changes to the global state, Piston carries out an analysis of system calls. Specifically, it checks for system calls that might not be repeatable. For example, if the interrupted and repeated invocation of the hijacked function both try to `unlink` the same file, an inconsistency between their executions will arise. Because Piston does not have a complete model of all system calls, it presents these lists to the analyst for review. If any system calls are deemed not repeatable, then the analyst must provide a rollback routine to undo the effects of the system calls.

4.4.3 Caller State Recovery

Regardless of whether the hijacked function has successfully run or needs to be restarted, the state of the caller function must be recovered. Although the general problem of restoring

registers and memory to the state of the execution before the overflow is undecidable, we have found that there is often enough data remaining to recover the original state. Our key insight is that, due to the way programmers write source code and compilers compile it, the stack frame and registers of a function often contain *redundant data*, which can be used to restore the corrupted data. In our case, this means that the value of a corrupted stack variable or register can often be determined as some equation of other stack or register values.

Listing 4.1: An example showing where stack variables have redundant information.

```
1 mov     eax, [ebp+var_14]
2 mov     edx, [ebp+var_8]
3 sub     eax, edx
4 mov     [ebp+var_3C], eax
5 call    hijacked_func()
```

Before delving into Piston's approach to state recovery, we provide and briefly discuss an example of such redundancy in Listing 4.1. Assume the programs instruction pointer is currently at line 5. The stack variable `var_3C` is redundant since it can be computed from the other stack variables, specifically, `var_3C = var_14 - var_8`. Thus, if the overflow clobbers `var_3C`, it can be recovered from `var_14` and `var_8`.

Data Filtering

Before recovering corrupted state, Piston must identify *what* state needs to be recovered. If the hijacked function completed successfully, we must restore any stack variables or registers that were corrupted by the exploit and will be used later in the caller function. Additionally, if the hijacked function was interrupted, we must also restore all of the arguments (on the stack and in registers) that are passed to the hijacked function.

As described in Section 4.4.1, Piston identifies the range of registers and stack variables that were clobbered by the exploit. In fact, not all of these values *must* be recovered. For example, if a callee-saved register is written to immediately after the hijacked function returns,

its value after exploitation, whether or not it was corrupted, is irrelevant, and there is no need to restore it. Piston identifies these cases by computing the control flow graph of the hijacked function and identifying accesses to stack variables and registers. Then a dependency analysis is run on the control flow graph to check if any path exists where a corrupted register or stack variable is read before it is overwritten. If no such path exists, Piston marks the register or stack variables as *unused* and filters it from further state recovery steps.

One caveat must be mentioned for stack values. In some cases, the caller function might pass a pointer to the stack as an argument to the hijacked function. Normally, this happens when a buffer or structure resides on the stack and must be used by the hijacked functions. If the hijacked function performs complex operations on this pointer (such as passing it into other functions or system calls), Piston's static analysis is unable to safely recover these effects. Piston makes the assumption that the passed-in pointer points to the *beginning* of the structure and assumes that the hijacked function may have corrupted anything on the stack after this pointer.

At the end of this step, Piston has a *recovery set* of the registers and stack values that must be recovered before the caller function can resume execution.

Data Recovery

Piston recovers state data by analyzing two locations in the caller function: the function prologue and the hijacked function call site.

Generally, functions will initialize several registers in the prologue and use them for the remainder of the function. This is especially true for registers such as the base pointer (i.e., `ebp` on x86), which are typically set at the beginning of a function. The values of registers that are set in this way can often be determined by analyzing the prologue of a function. Likewise, the caller function *prepares* the call-site of the hijacked function by copying its arguments into argument stack variables and registers. Most of the time, these arguments are passed by value

and are drawn from other parts of the state, creating data redundancy that can be leveraged to restore their values when they are corrupted by the exploit.

To avoid under-approximations, Piston does not reuse the exploitation trace in the data recovery step. The control-flow path from the trace may differ from the one that will be executed on the remote server. For example, the remote server may have internal state such as a linked list, which will result in a different control flow than the one in the concrete trace.

To recover data, Piston will analyze two locations with symbolic execution. The first is the start of the caller function up until the first branch. The second location is the callsite of the hijacked function, starting at the earliest basic block from which there is only one path that reaches the call.

Piston analyzes these locations with under-constrained symbolic execution and extracts the relationships between data that must be recovered and the uncorrupted data currently existing in the state. We represent these relationships as equations that produce the recovered values of corrupted data when provided the values of the uncorrupted data. These equations are then examined to verify that all values in the recovery set can be recovered from existing data in the stack.

For example, in Listing 4.1, when Piston symbolically analyzes the callsite of the hijacked function it will generate a constraint that `var_3C = var_14 - var_8`. If Piston determines that `var_3C` will be overwritten, but not `var_14` or `var_8` then it will determine that `var_3C` is recoverable.

If all values in the recovery set can be recovered from existing data on the stack, Piston saves this set of equations as the *repair routine*. Otherwise, Piston requires the analyst to provide a partial repair routine that recovers corrupted values that are still missing. The repair routine will be executed after the remote process is patched and before it is restarted, as explained in Section 4.5.

4.5 Remote Patching

Until this point, Piston's analysis has been offline: no connection to the remote process has been made. This section describes how Piston uses the provided exploit specification to achieve code execution in the remote process, and applies the results of the offline analyses to repair, patch, and resume the remote process.

The astute reader will recall that, in the previous analyses, Piston recovered the following information for use during the remote patching:

Patch set. In Section 4.3, we described how Piston identified the set of patches to apply to the remote process to turn it into a functional copy of the replacement binary.

Rollback routine. We introduced in Section 4.4.2 Piston's strategy for undoing the effects of the hijacked function, if it is determined to have been interrupted by the exploit.

Repair routine. Piston's approach to creating a routine to repair the remote process state after exploitation is detailed in Section 4.4.3.

While generating this information is complex, the rest of the process is straightforward. Piston executes the following steps, in order:

1. First, Piston launches the exploit against the remote process. The exploit hijacks the control flow of the remote process and loads a first-stage payload, provided by Piston, which facilitates the execution of the rest of the repair and patching tasks. We call this payload the *patching stub*.

2. Next, Piston transfers the repair routine to the patching stub. The patching stub executes the repair routine to repair the damage done by the exploit to the remote process state.

3. If, during the prior offline analysis, Piston determined that the exploit caused an interruption of the hijacked function (i.e., it did not terminate successfully), Piston transfers the rollback

routine to its patching stub and executes it to undo the effects of the hijacked function. As discussed in Section 4.4.2, in this case, the hijacked function will be restarted after the patching process is complete.

4. Piston transfers the patch set to the patching stub. The patching stub applies this patch set to the remote process, transforming it into a program that is functionally equivalent to the replacement binary.

5. Finally, the patching stub returns control to the remote process. If the hijacked function completed successfully, it simply returns to the instruction, inside the caller function, after the call to the hijacked function. Otherwise, control flow returns to the beginning of the hijacked function.

After these steps are completed the remote process has been hot-patched. The remote host is now effectively running the replacement binary, and this has been done without restarting the entire process or performing any permanent changes.

The rest of this section will discuss other minor points relating to Piston's remote patching step.

4.5.1 Exploit Requirements

Piston has very simple requirements for the provided exploit specification. In short, the specification must describe an exploit that achieves code execution and loads Piston's patching stub. As discussed throughout the paper, if this exploit uses a stack-based buffer overflow to achieve code execution, Piston can often carry out the rest of its work automatically. Otherwise, the user must also provide the rollback and repair routines.

4.5.2 Optional Patch Testing

Piston supports an optional patch testing step between the offline analyses and the actual remote patching described earlier in this section. If the analyst provides a *test case* to verify that the process has been properly patched, Piston carries out a test run against a locally-executed copy of the original binary. After patching this local process, Piston verifies that the test case passes when run against it. While this is a very straightforward concept, we found that it greatly eased cases when rollback and repair functions had to be provided manually by the user.

4.5.3 Persistence

Piston is meant to patch the running process *ephemerally* (i.e., without making any actual changes to the filesystem or firmware). While Piston can, during the patching process, execute a user-provided *persistence routine* to persist its changes (for example, by overwriting the original binary on disk), this is not Piston's standard use-case. In fact, we expect that, generally, the process that Piston patches will not have the proper access to write to its original binary on-disk. For example, server processes on Linux almost never have write permissions to their own binaries, and Piston would be running with the same permissions as the server process while patching it.

To patch forking services, Piston would need to apply the patch to the parent process. There are no theoretical limitations which prevent Piston from attaching to, and patching, a parent of the exploited process, granted that our exploited process has permissions to attach to a parent and in addition, that the underlying operating system supports process tracing.

Ephemeral patching itself is a very powerful technique, even without the ability to commit the changes to disk. In Section 4.6, we showcase how to quickly patch a security flaw in a web server to which the analyst may not have access. That application of Piston does not need to be persistent to be useful. Furthermore, other hot-patching systems such as PatchDroid choose

to only patch ephemerally [70].

4.6 Evaluation

We evaluate Piston in two ways. First, we test its ability to recover the program state after a stack buffer overflow on all of the applicable binaries from the Cyber Grand Challenge Qualifying Event (CQE). For all CQE binaries with stack buffer overflows, we test if Piston can recover enough state in the caller function, such that the state can be completely restored after an exploit achieves arbitrary code execution. Then, we test Piston's patching functionality on five of those binaries as well as a real-world binary, NGINX 1.4.0 (which is vulnerable to CVE-2013-2028) by creating exploits and using Piston to apply the patch, recover state, and resume execution.

4.6.1 Dataset

We chose targets for Piston that would allow us evaluate Piston's state recovery methodology. We use binaries from the Cyber Grand Challenge because these represent a large number of binaries containing a wide variety of functionality. Additionally, CGC binaries are guaranteed to have at least one vulnerability as well as a Proof Of Vulnerability (POV) which causes it to crash. As such, these targets are used to test Piston's recovery capabilities in a wide range of binaries. We took the 126 single-binary applications from the CQE and discarded any which did not crash with the provided POV in our testing environment leaving us with 102 binaries. Of those 102, we found that 24 crashed due to an inter-frame stack overflow. We use all 24 for testing Piston's recovery capabilities.

To test the end-to-end patching and recovery from an exploit, we chose five binaries from the above set. For each of these binaries we had to write an exploit which would give us arbitrary code execution. This was required because the provided POVs only lead to crashes,

many of which do not crash with control of the instruction pointer.

Along with the CGC binaries, we chose NGINX 1.4.0, which is vulnerable to CVE-2013-2028, to test Piston on a real-world application. NGINX is proves to be an interesting candidate due to its unique architecture among webservers: it initializes a fixed number of worker processes that persist throughout the entirety of the server's uptime. This allows us to patch the individual workers of the NGINX server by repeatedly connecting to the server.

4.6.2 Recovery Results

To test Piston's recovery capabilities we used the 24 CQE binaries containing an inter-frame stack overflow. We constructed two patching stubs, one that relies on the absence of NX (shellcode stub), and one that bypasses NX using return oriented programming (ROP stub). The shellcode stub is 23 bytes in length whereas the ROP stub is 40 bytes. We trace each of those binaries with their accompanying POVs and use Piston's built-in functionality to identify the exploitation point and the hijacked function in which the overflow occurs. Then we set the overflow amount to that which is needed for each of the patching stubs and check if piston can recover the state.

Piston was able to correctly identify the corruption point in all cases, and was thus able to identify the corrupted data. For the shellcode stub, Piston was able to completely recover the corrupted data for 22 out of 24 binaries. For the ROP stub, which clobbers more bytes of the stack, Piston was able to completely recover the corrupted data for 20 out of 24.

4.6.3 End-To-End Results

Piston was able to patch all five binaries from our CGC end-to-end dataset as well as patch NGINX, with only two of these six binaries requiring input from the analyst. Only one of these binaries, CROMU_00038, required the analyst to write code. In the other one that required

Binary Name	Function Interrupted?	Fully Automated Rollback?	Fully Automated Repair?	Caller Stack Bytes Recoverable
CROMU_00017	Yes	Yes	Yes	144
CROMU_00020	Yes	Yes	Yes	52
CROMU_00037	No	N/A	Yes	4
CROMU_00038	Yes	Yes	No	4
CROMU_00039	Yes	Yes	Yes	303
NGINX	Yes	No	Yes	28

Table 4.1: Breakdown of patches from Piston

input, NGINX, Piston was unable to generate a rollback function, but the analyst was able to quickly determine that no rollback function was actually necessary.

In the one CGC binary that required the analyst to write code, Piston's patch testing step reported a possible problem. Upon inspection, we discovered that the patched binary sanitizes the input before control reaches the hijacked function, but the runtime patch was restarting the hijacked function with the unsanitized input. By providing a repair function that sanitized the input in memory, the patching was able to proceed as expected.

As part of our experiments we evaluated how much stack space in the caller function could be overwritten before Piston would need to undo and restart the caller function as well. We iteratively increased the amount of overflow until Piston reported that the caller function's frame could not be recovered. These results are shown in Table 4.1. We found that there was a large variation in the number of bytes in the caller's frame that were recoverable; the results ranged from only four bytes to over three hundred.

4.6.4 NGINX Patching

In July of 2013, both NGINX version 1.3.9 and 1.4.0 were found to be vulnerable to a stack-based buffer overflow which results from improper handling of HTTP chunked transfer-encoding (this vulnerability was given the label CVE-2013-2028). NGINX is not a simple binary; the source code alone for this version approaches 180,000 lines of code. By successfully

patching NGINX through this CVE, we demonstrate Piston's effectiveness and applicability.

We began our evaluation by compiling two versions of NGINX; one version represents the original binary, and the other is the replacement binary. We obtained the original binary by downloading the NGINX 1.4.0 source code and compiling it. For the replacement binary we took the same source code and applied the CVE-2013-2028 patch file provided by nginx.com [71]. Next we developed an exploit specification targeting the vulnerability. Our exploit specification is simply an exploit script which gets to shellcode execution on an NGINX worker process; many exploits for this particular CVE can be found online [72, 73].

While Piston was analyzing the hijacked function, it determined that the function was interrupted and would need to be repeated. Upon determining that the hijacked function must be repeated, Piston identified small changes which would be made to the global state of the process on a repeated call of the function. Piston was unable to generate a rollback routine for these particular changes, so deferred the creation of a rollback routine to the analyst, highlighting the changes made during the repeat. In a matter of seconds, we, as analysts, can see that the effects of a repeat call are inconsequential, and inform Piston to carry on without rollback.

Next, Piston was able to successfully determine that four bytes of the caller's state were destroyed, as a result Piston then generated a repair routine which recovered these four bytes. However, for the sake of evaluation, we show that 28 bytes of the caller's state could have been corrupted without hindering Piston's ability to generate a repair routine automatically.

After these steps, the brunt of the analysis is complete. Piston now executes the patcher using the exploit specification provided to first get shellcode execution. With shellcode execution Piston then reads in and executes the repair routine generated earlier. Then, Piston's shellcode performs the patching process and soon reports that the patching is complete.

We verify that the NGINX web server is still running by manually making a request with a browser. Next, we verify that the server has successfully been patched by attempting again to exploit the server, but this time attempting to redirect control flow to an invalid address. After

this exploit attempt, we again make a request to the web server with a browser and verify that NGINX has withstood crashing (we configured NGINX to use a single worker, so a crash in a single worker would have resulted in the entire server being inoperable).

4.7 Limitations

One primary limitation of Piston is that the fully automated recovery steps only succeed on stack-based buffer overflows. For other types of corruption, an analyst typically needs to examine the data which was identified as corrupted, and then decide how it can be recovered. The reason for this limitation is that although the Data Filtering and Data Recovery in Section 4.4.3 can be thought of in generalized steps, they do not produce adequate results when applied to data outside of the stack.

Data Filtering. On the stack frame, we have the advantage of detecting which instructions access stack variables, whereas for data in the heap, due to limitations in the current state of static analysis, it is rare to know which instructions will read or write from a specific object. Some thorough type analyses [66, 67] may be able to identify accesses to objects of the same type, but cannot identify if those accesses are to the same object which was corrupted. Data filtering of heap corruption might require a *semantic* understanding of the program. Such understanding is outside the reach of current techniques.

Data Recovery. Data recovery requires data redundancy. That is, we must be able to automatically deduce the value of data from other values in memory or registers. In the case of stack data, we showed how other values can provide this redundancy in Section 4.4.3. However, if we consider corruption to heap or globals, one problem is that the data is typically created at an earlier point in program execution, often in a stack frame which has since been discarded. Unless we still have the stack frame in which a heap object was initialized, we are unlikely

to have data which provides the necessary redundancy to recover the object.

However, there *are* cases when Piston can be applied to vulnerabilities other than stack overflows.

Here, we describe one such case, in which Piston was able to automatically patch a binary using a heap overflow vulnerability. The CGC binary NRFIN_00004, which was not included in our testing dataset because it does not have a stack-based buffer overflow, contains an intra-object heap overflow. The heap object contains a string followed by several function pointers. When the string overflows, the function pointers are overwritten, and another command handler will call an overwritten pointer.

We began by designing a custom heuristic to Piston to detect the corruption point. The heuristic was that for heap objects, any pointer to a function cannot be changed to point at an address that is not the beginning of a function. With this heuristic, Piston correctly identifies that the two function pointers in the heap object were corrupted. From there, Piston follows its normal mode of operation: it injects the patching stub into the binary, executes it, replaces all functions in the patch set, then restarts the execution of the hijacked function, which previously contained the heap overflow. Piston's underconstrained symbolic execution can detect that the corrupted pointers will be overwritten by the restarted (and patched) hijacked function, so no data needs to be recovered, avoiding the problem of the lack of data redundancy.

This is not a *general* application of Piston to heap overflows, so we include it here as opposed to the core approach discussion. However, it demonstrates that, with minor manual work, Piston can be adapted to a wider range of vulnerabilities. In this case, it only required a different corruption point detection heuristic.

4.8 Conclusion

In this paper, we presented Piston, the first proposed approach for remote hot-patching of uncooperative processes. Piston patches processes through exploitation, allowing us to patch software which was originally considered unpatchable. Piston makes the novel contribution of exploitation clean up, recovering from many of the unpredictable state changes introduced during a memory corruption exploit. We evaluated Piston on a large, real-world binary and a synthetic dataset provided by DARPA. Piston was able to apply patches to each binary and, in most cases, carried out the patch completely automatically.

Chapter 5

Exploring Abstraction Functions in Fuzzing

5.1 Introduction

As our society becomes increasingly dependent on software, the security of this software becomes paramount. Whereas, at one time, security could be approached *reactively*—responding to vulnerabilities only after their exploitation by attackers—this is no longer acceptable. Modern security is *proactive*, with researchers attempting to identify and fix software flaws *before* they can be found by attackers.

One method, fuzz testing, has emerged as the preeminent automated security analysis technique in the real world. This technique has impressive results thus far. A modern example is American Fuzzy Lop (AFL) [29], a powerful fuzzer based on genetic techniques, which is responsible for the detection of hundreds of real-world flaws, including high-impact vulnerabilities such as the Stage-fright vulnerability [74]. AFL's success has spawned a veritable "cottage industry" of researchers looking to improve various stages of the fuzzing process. However, without a formal, scientific model on which to base these improvements, the field of

fuzzing was explored in an extremely *ad hoc* way, and it is difficult to understand the relative merit of different approaches.

In this paper, we present the first formalization of input evaluation and selection in fuzzing, borrowing concepts from the field of static analysis. Formally, a fuzzer generates input test-cases and dispatches them to a program, dynamically triggering a subset of the potential states that the program can reach. However, the full set of states is potentially infinite. For tractability, the formal fuzzing process uses an approach-specific *abstraction function* to reduce this set to an *abstract state space*, allowing the fuzzer to identify "promising" test-cases for further mutation by selecting test-cases that correlate to different abstract states in the state space.

We explore the implications of our formalization-derived observation on the effectiveness of evolutionary fuzzing techniques in the second half of the paper, and we show that the application of different abstraction functions, and the use of multiple abstraction functions in tandem, shows promise for improving state-of-the-art fuzzing techniques. To maximize the benefit of this work to the community, we will open-source the resulting tool upon publication.

In summary, this paper makes the following contributions:

- To form a scientific base for research into future approaches in fuzzing, we provide a formalization of the input evaluation and selection process in fuzzing and redefine current work in the context of this formalism.

- Stemming directly from an observation made during the formalization process, we propose a diversification of fuzzer abstraction functions and design a number of such functions that can be used both alone and in composition with each other.

- We implement an extensible framework for the development and evaluation of fuzzer abstraction functions, evaluate its impact on the effectiveness of a modern fuzzer, and open-source our work for reproducibility and for the community to build upon.

5.2 Formalizing

Fundamentally, the goal of a fuzzer is to find in a given program software bugs that violate the security properties of the program. As fuzzing is a dynamic technique, the fuzzer finds bugs by providing input to the program in an attempt to trigger a program state that violates a security specification (for example, accessing invalid memory). Fuzzing can be viewed as an iterative process, targeted to explore the state space of a given program *completely*. Unfortunately, exploring the entire state space of a program is equivalent to solving the halting problem, which is undecidable.

Similar to program testing, fuzzing is an automated testing technique. It tries to generate *interesting* inputs as fast as possible within a given resource budget.

5.2.1 Concepts

First, we define some notions that will be used throughout our fuzzing formalism:

Input: Programs consume input data to drive their operation. An *input*, ι is our representation of this input data. Note that this notion of input covers all types of input to a program (command line, network, files etc).

Input Space: The alphabet of possible inputs is Σ and the set of all possible inputs is Σ^*. If the length of the input is not bounded, the set Σ^* is infinite.

Concrete State: The snapshot of all the processor registers, the program's memory, file system operations, or anything else that effects the operation of the program represents the concrete state, r, of a program. The symbol C indicates the set of all the possible concrete states of a program.

Concrete State Space: Each input ι triggers a series of concrete states as it is processed. The

trace of all of the concrete states reached by the input ι, denoted as cs_ι, is the Concrete State Trace of the input.

Because there is a potentially infinite amount of inputs that could be read by the program, the set of all Concrete State Traces can be infinite. CS denotes the set of all concrete state traces for a program.

Abstract State Space, AS: As the set of Concrete State Traces CS can be infinite (or computationally infeasible to enumerate), fuzzing techniques must abstract the concrete state space so that different states (and therefore, different inputs) can be considered equivalent. Here we derive inspiration from Abstract Interpretation [75], and similar to abstract domains in Abstract Interpretation we define the *Abstract State Space* (AS) as a domain to which a concrete state space will be mapped to. The elements of this domain are called *abstract states*.

These concepts will be used as the basis of a formal definition of fuzzing.

5.2.2 Mapping to Abstract States

Fuzzing techniques reason over the abstract state space of an input instead of the concrete state space. This allows them to group inputs having different cs enabling efficient generation of interesting inputs.

The mapping between AS and CS is handled by two functions:

Abstraction function (α): This function maps a concrete state trace to an abstract state.

Formally, $\alpha : CS \rightarrow AS$ and A is a set of abstraction functions.

Concretization function (γ): This function maps an abstract state to a list of concrete states.

Formally, $\gamma : AS \rightarrow CS$.

For a given input ι and a corresponding cs_ι, we can compute the corresponding abstract state by applying α as $\alpha(cs_\iota)$. We call this the *Input Abstract State (IAS)*.

Formally, $IAS(\iota, \alpha) \ = \ \alpha(cs_\iota)$.

The tuple of an input (ι) and corresponding IAS form the ***Fuzzing Result*** (FR) of the input ι.

Formally, $FR(\iota, \alpha) \ = \ (\iota, IAS(\iota, \alpha))$.

For a set of inputs I and an abstraction function α, the set of corresponding fuzzing results are called ***Fuzzing Results Set*** (FRS).

Formally, $FRS(I, \alpha) \ = \ \{FR(\iota, \alpha), .. \mid \forall \iota \in I\}$.

For a set of inputs I and a set of abstraction functions A, we can define **Complete Fuzzing Results** (CFR) formally as:

$CFR(I, A) \ = \ \{FR(\iota, \alpha), .. \mid \forall \alpha \in A, \iota \in I\}$.

For a given set of abstraction functions A, two inputs ι_1 and ι_2 are considered the same iff $CFR(\{\iota_1\}, A) \ = \ CFR(\{\iota_2\}, A)$.

To shorten the notation we will use α (ι) as the abstraction of the Concrete State Trace triggered by ι, i.e. $\alpha(\iota) := \alpha(cs_\iota)$.

5.2.3 Fuzzing Techniques and Procedures

With these notions defined, we can formally define a fuzzing technique ($\hat{F}$) as a function that takes the following inputs:

- The set of abstraction functions to be used for the current iteration (A_{curr}).
- The program p_{curr} to be tested, with additional instrumentation as needed by the abstraction functions A_{curr}.
- The set of complete fuzzing results of all previously tested inputs and abstraction functions (CFR_{prev}).

- A set of inputs to be used for current iteration (I_{curr}).
- The time and resource consumed thus far ($t\ r_{curr}$).

and produces the following outputs:

- A set of inputs to be used for the next iteration (I_{next}).
- A set of abstraction functions to be used for the next iteration (A_{next}).
- A version of the program p_{next}, with additional instrumentation needed by the abstraction functions A_{next}.
- The new complete fuzzing result set, which includes the complete fuzzing result of I_{curr}. i.e., $\{CFR_{prev} \cup CFR(I_{curr}, A_{curr})\}$.
- The new time and resource consumption ($t\ r_{next}$).

Formally, a fuzzing technique can be defined as:

$$\hat{F} : (p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all})$$
$$X\ (p_{all}, I_{all}, P(A), P(CFR_{all}), t_{all}, r_{all})$$

where p_{all} is the set of all possible functionally identical copies of the program p, $P(A)$ is the power set of all possible abstraction functions, $P(CFR_{all})$ is the power set of complete fuzzing results across all possible inputs I_{all} and all possible sets of abstraction functions A (formally, $CFR_{all} = CFR(I_{all}, A)$), t_{all} is the set of all possible time consumptions, and r_{all} is the set of all possible resource consumptions.

In every iteration, a fuzzing procedure usually stores all the interesting inputs used during the iteration. An input is *interesting* if it explored an abstract state that is *not* reached by any of the previous inputs. Formally, let the set of all interesting inputs stored by a fuzzing technique

be $\widetilde{I}$, an input ι in the iteration, i.e., $\iota \in I_{curr}$, can be considered *interesting* if the following relation holds:

$$\exists \alpha \in A_{curr} \mid \alpha(\iota) \not\sqsubseteq_{\alpha} \bigcup_{i \in \widetilde{I}} \alpha(i)$$

where A_{curr} is the set of abstraction functions used in the iteration.

The above relation ensures that there is an abstraction function, according to which the input ι explored an abstract state that is not reached by any of the previously used inputs.

The *fuzzing process* $\bar{F}$ applies the fuzzing technique $\hat{F}$ iteratively until the resource budgets are consumed.

A fuzzing process $\bar{F}$ is given the program p, an initial set of inputs I_{init} (known colloquially as "seeds"), a set of initial abstraction functions A_{init}, a fuzzing technique $\hat{F}$, a time budget t, and a resource budget r. A fuzzing process will return the set inputs that trigger invalid program states $I_{invalid}$.

$$\bar{F} : (p_{all}, I_{all}, \hat{F}_{all}, t_{all}, r_{all}) \; X \; I_{all} \tag{5.1}$$

The fuzzing process $\bar{F}$ applies $\hat{F}$ iteratively until the provided resource budgets are exhausted.

$$... \; \hat{F} \circ \hat{F} \circ \hat{F} \circ \hat{F}(p, I_{init}, A_{init}, \emptyset, 0, 0) \tag{5.2}$$

5.2.4 Fuzzing parameters

Given this formal definition of fuzzing, *any* fuzzing technique can be described by defining the following parameters:

Input Generation (I_{next}): This describes how the new input is generated, i.e., the technique used to generate I_{next} for an iteration of the fuzzing technique. An intelligent input generation is an innate feature of any fuzzing technique. There are different ways to generate inputs, whether by using random data, by using the fuzzing result of the previous inputs,

or a combination of both.

Abstraction Functions (A): This is the total set of abstraction functions that could be used by the fuzzing process.

Abstraction Function Selection (A_{next}): Similar to input generation, this parameter describes the mechanism used to select the abstraction functions to be used for the next iteration. Most of the fuzzing techniques have a single abstraction function (i.e., $|A| = 1$) and use the same abstraction functions in every iteration (i.e., $A_{next} = A_{prev}$). We call these techniques *Single Abstraction Fuzzing (SAF).*

To demonstrate the generality of our model, let us define some of the existing fuzzing technique using these parameters.

AFL [29]: For input generation, AFL uses various mutation of interesting inputs such as: bit-flipping, byte-flipping, splicing, etc. It is a SAF technique with the following abstraction function:

$$\alpha_{afl}(\iota) \;=\; \{(bb_i^1, bb_j^1, log_2(n_1)), (bb_i^2, bb_j^2, log_2(n_2)), ..\}$$

where $(bb_i^*, bb_j^*, log_2(n_*))$ are pairs of basic blocks (bb) such that bb_j^* is visited right after bb_i^* for n_* number of times when the program processed the input ι and log_2 is the logarithm with base 2.

Dowser [76]: For input generation, Dowser uses constraint solving to generate interesting inputs.

It is also an SAF technique with a slightly different abstraction function than SAGE: instead of collecting all the constraints, it only collects the constraints at predetermined

program points P_{Dowser}.

$$\alpha_{Dowser}(\iota) \;=\; \langle (c^1, b), (c^2, b), (c^3, b), ... \rangle \tag{5.3}$$

where $c^* \in P_{Dowser}$ is a conditional statement of the program p reached by the input ι and b is the Boolean value (i.e., $b \in \{1, 0\}$), that indicates the result of the conditional statement.

Driller [33]: It uses both AFL method and constraint solving (similar to SAGE) for input generation. It is a SAF technique with the AFL's abstraction function i.e., α_{afl}.

VUzzer [19]: It generates inputs based on mutation and combinations of interesting inputs.

It is also a SAF technique with the following abstraction function, which is based on a scoring function ($score$) that is based on the basic blocks reached by the input.

$$\alpha_{VUzzer}(\iota) \;=\; \{(len(\iota), score(bb^1_w, bb^2_w, ...))\}$$

where $len(\iota)$ is the length of the input in bytes and bb^*_w is a pre-calculated weight of the basic block bb^* reached by the input ι.

Steelix [20]: This is a technique customized to fuzz magic byte based programs. For input generation, Steelix uses conditional mutation of the input bytes where a comparison failed. Steelix is actually a multi-abstraction fuzzer that combines AFL and the results of interesting comparison operations. Formally,

$$\begin{aligned}\alpha_{steel}(\iota) \;=\; \{&(bb^1_i, bb^1_j, lg_2(n_1)), (bb^2_i, bb^2_j, lg_2(n_2)), ...,\\ &(c^1, n_1), (c^2, n_2), (c^3, n_3), ...\}\end{aligned}$$

Here, the first part is similar to AFL, where $(bb_i^*, bb_j^*, log_2(n_*))$ are pairs of basic blocks (bb) such that bb_j^* is visited right after bb_i^* for n_* number of times.

The second part captures the results of interesting comparisons: where c^* is a conditional statement reached by the input and n_* is the number of bytes in the conditional statement that matched.

Angora [21]: Angora uses a single abstraction function similar to AFL's, with the addition of context sensitivity given by taking a hash of the callstack. Angora also generates new inputs using byte level taint tracking and a gradient decent algorithm for trying to satisfy conditional statements.

$$\begin{aligned}\alpha_{angora}(\iota) \;=\; &\{(bb_i^1, bb_j^1, h(stack_1), log_2(n_1)),\\ &(bb_i^2, bb_j^2, h(stack_2), log_2(n_2)), ..\}\end{aligned}$$

As shown, the provided formal model of fuzzing helps in understanding various fuzzing techniques in a systematic manner. Any fuzzing technique can be easily defined using our Fuzzing Parameters ($Section$ 5.2.4).

By describing existing fuzzing techniques using our formal model, one can see the following observations emerge:

> *Observation 1*: Most current fuzzing techniques either develop new input generation techniques (e.g., Driller [33], DIFUZE [77]) or change, in tandem, both the abstraction function and the input generation technique (e.g., VUzzer [19], Angora [21], and Dowser [76]). In actuality, these two concepts are orthogonal.

Observation 2: Most existing fuzzing techniques are *Single-Abstraction Fuzzers* (SAF). That is, they use the same single abstraction function in every fuzzing iteration. However, there is no obvious reason why this *must* hold for all techniques.

This leads us to a natural research direction:

- With a fixed input generation technique, how does the chosen abstraction functions affect the effectiveness of fuzzing?
- Can a fuzzing strategy use *multiple* abstraction functions? Will it be more effective (given the same budget) than the corresponding SAF variation?

In the next section, we will describe a number of alternate abstraction functions that we will use to explore this direction of research.

5.3 Abstraction Functions Explored

There are potentially infinite ways to abstract the concrete state space covered by an input on a program. Some aspects that abstraction functions could be based on are: (1) Code coverage, the set of basic blocks accessed [78], (2) Data access, the set of all global variables accessed, or (3) Function invocations, is the set of all library function called during program execution.

As mentioned in $Section\ 5.2.4$, there could be several possible abstraction functions, where each could be effective in exploring a particular concrete state space of the program. Based on the requirements of effective exploration and performance overhead, we implemented six different abstraction functions, three of which are similar and have fine-grained granularity and the remaining three attempt to abstract different concrete state spaces of the program.

***Basic Blocks* Abstraction (α_{bb})**

This is the most basic abstraction function which simply tracks the number of times each basic block was executed by the program. Instead of maintaining the raw counts, we use logarithmic counting set (as used in AFL [29]). Formally:

$$\alpha_{bb}(\iota) = \{(bb_i, log_2(c_i)), (bb_j, log_2(c_j), ...\}$$

Where, bb_* is the basic-block executed and c_* is the number of times corresponding basic block is executed when the program processed the input ι.

The intuition behind this abstraction is to capture the basic-block coverage achieved by an input on the program, under the assumption that more coverage yields more bugs.

***Edges* Abstraction (α_{edge})**

This abstraction function increases the granularity of *Basic Blocks* abstraction by tracking the number of times (using logarithmic counting) an *edge* between basic blocks is executed. This is the same abstraction ($Section$ $5.2.4$) used by the AFL fuzzer [29].

***Block Triples* abstraction (α_{triple})**

The *Block Triples* abstraction function is similar to the *Edges* abstraction function, however instead of tracking edges (which is a pair of basic blocks), here we track all of the three consecutive basic blocks visited during the execution of the program.

The intuition here is that because the edges abstraction is successful (as shown by AFL), then perhaps increasing the granularity of the abstraction could increase its effectiveness.

```
int check_header(char *data, char *header) {
  if (data[0] == header[0] && data[1] == header[1]
      && data[2] == header[2])
    return 1;
  else;
    return 0;
  }
}
int handle_data(char *data) {
  if (check_header(data, "TXT")) {
     // safe code
  }
  else if (check_header(data, "PDF")) {
     // BUG
  }
}
```

Listing 1: In this example, a utility function, `check_header` is used for multiple checks. An abstraction function that only tries to cover all edges will likely fail to pass both checks, because it will have already seen the edges within the utility function.

Edge + Return Loc **Abstraction** ($\alpha_{edgeret}$)

This abstraction, in addition to the *Edges* abstraction, also considers the calling function. In static-analysis terms, this is a 1-context sensitive version of the *Edges* abstraction. Formally,

$$\alpha_{edgeret}(\iota) = \{(bb_i^1, bb_j^1, ret_1, log_2(n_1)), (bb_i^2, bb_j^2, ret_2, log_2(n_2)), ..\}$$

Where ret_* is the calling context under which the corresponding edge (bb_i^*, bb_j^*) was visited. The rest of the terms are the same as the *Edges* abstraction.

The intuition behind this abstraction is that if there is some function that performs a comparison, such as a `strcmp`, it is useful to satisfy that comparison when it is called in different contexts and not in a single context. For example, in Listing 1 a single function is used to check a three-byte header. A *Basic Blocks* or *Edges* abstraction can satisfy the check once because, as each successive byte is matched, the new input will be considered interesting. However, it will only match one header, because for a second header those blocks/edges will have already been seen in the utility function, and it would not be able to successively match the new bytes. On the other hand, the *Edge + Return Loc* abstraction will match the header multiple times, because it *includes the calling context.*

Function Context **Abstraction** ($\alpha_{context}$)

In this abstraction function, we attempt to capture the context of the executed functions. As the context could be potentially unlimited, we limit the length of the context to four. We capture the context as the sequence of the last four return address on the call-stack at the entry of each executed function. In the case where the call-stack has less than four return addresses we use *null* instead. This is done by xoring the callstack entries.

The motivation for this abstraction is that, in many real-world vulnerabilities, the context of certain function invocations is critical. For example, in a JavaScript engine, the JavaScript code can add or remove elements in an array. However, these operations might not be safe if the code is called from inside a `sort` function which does not handle a changing array size (as in CVE-2013-0997 [79]).

Method Calls **Abstraction**(α_{omcp})

This abstraction function is specialized for object-oriented programs, and we capture the pairs of methods executed on the same object instantiation. Consider an example, where we have an object `foo` with methods `A()`, `B()`, and `C()`. If the execution of a program given input ι results in the following method invocation sequence: `foo.A()`, `foo.B()`, `foo.C()`, then we will add the pairs `A`-`B`, `A`-`C`, `B`-`C`, to the counts. If there is a second object `bar` that is the same class as `foo` and the program execution is: `foo.A()`, `bar.B()`, `foo.C()`, `bar.B()` we will add the pairs `A`-`C`, `B`-`B` to the counts because the method calls are tracked on a per-object basis.

Steelix ($\alpha_{steelix}$)

Although this is actually a multi-abstraction as explained in Section 5.2.4, it is included here as it is one of the strategies we explore in our evaluation. The details are previously

explained, but briefly, *Steelix* uses two abstraction functions, edges, and one which looks at comparisons with important values.

5.3.1 Multi-Abstraction

As demonstrated in Section 5.2.4, most existing fuzzing techniques are Single Abstraction Fuzzing (SAF). In this paper, we explore combining multiple abstraction functions so that they can provide new inputs to each other and combine the strengths that each abstraction function provides. For example, let us consider how the *Edges* and *Method Calls* abstractions described previously might pair well together. The *Method Calls* abstraction instruments method calls on objects and will consider different arrangements of method calls interesting, however it might not be able to trigger a particular method call in the first place. The *Edges* abstraction might easily find that new method call, however it may not find the different arrangements of the method calls interesting.

To combine abstractions, we chose to run them as parallel fuzzers and share inputs between them, similar to the ensemble fuzzing strategy presented by Chen et. al [80] and Wang et. al [81]. In other words, each fuzzer will use its own abstraction function but consider all interesting inputs that all fuzzers discover. Of course there are other ways of using multiple abstraction functions, such as switching between them, although exploring such other ways of combining them is out of scope of this paper.

5.4 Implementation

We implemented each of the abstraction functions including the Mult-Strategy abstraction using American Fuzzy Lop, as it is one of the most popular and effective open source fuzzers. To implement the Multi-Strategy abstractions we run each input selection strategy in parallel with separate (distinct) fuzzers, however the found interesting inputs are shared between each

fuzzing instance.

5.5 Evaluation

Lacking the formalism contributed by this paper, existing approaches in fuzzing do not maintain a separation between the abstraction function utilized and other details of the respective approaches. Additionally, it seems that current fuzzing techniques do not tend to leverage *multiple* abstraction functions. In this section, we explore both of these oversights, evaluating the effectiveness of alternate abstraction functions as well as the combination of multiple abstraction functions. We attempt to answer the following research questions.

RQ1: Does the choice of abstraction function affect the bug finding capabilities?

RQ2: How effective is it to combine different abstraction functions?

When evaluating these research questions, it is important to keep in mind we are not only looking to see if the choice of abstraction functions or combination thereof results in more crashes, but also to see if the bugs that are found differ. That is, if one abstraction function finds less bugs, but finds bugs not found by another than it is still interesting and would be worth applying.

Also, it is important to reiterate, as stated in Observation 2 in Section 5.2, that the choice of abstraction function (or functions) is *orthogonal* to other aspects of the fuzzing process. That is, though symbolically-assisted approaches (such as Driller [33]) would have a scaling effect on the numbers reported in this section, they would not effect the *relations* between these numbers. As such, we evaluate our system using a modification of the American Fuzzy Lop fuzzer.

Table 5.1: Multistrategy Fuzzer Configurations.

Selection Strategy	Abstraction Functions
Multi-Strategy1	*Basic Blocks*, *Edges*, *Block Triples*, *Edge + Return Loc*, *Function Context*, *Method Calls*
Multi-Strategy2	*Steelix*, *Basic Blocks*, *Edge + Return Loc*, *Function Context*

Table 5.2: This table shows the percentage of binaries that are crashed by the configuration in the corresponding row that are *also* crashed when using the configuration in the corresponding column. The value of each cell at row m and column n is: $\frac{C_{AF_m} \cap C_{AF_n}}{C_{AF_m}} * 100$, where C_k is the total binaries crashed when using the abstraction function k, AF_m, and AF_n are the abstraction functions of row m and column n respectively. The cells in the table are shaded based on the value veryhigh 100-95, high 95–90, medium 90–80, and low < 80.

	Basic Blocks	*Edges*	*Block Triples*	*Edge + Return Loc*	*Function Context*	*Method Calls*	*Multi-Strategy1*	*Steelix*	*Multi-Strategy2*
Basic Blocks	—-	94.51	94.51	96.70	80.22	61.54	96.70	94.51	95.60
Edges	94.51	—-	94.51	96.70	81.32	63.74	96.70	97.80	96.70
Block Triples	93.48	93.48	—-	97.83	80.43	63.04	98.91	97.83	97.83
Edge+Ret Loc	87.13	87.13	89.11	—-	74.26	58.42	99.01	94.06	98.02
Function Context	93.59	94.87	94.87	96.15	—-	73.08	96.15	100.00	100.00
Method Calls	91.80	95.08	95.08	96.72	93.44	—-	98.36	98.36	98.36
Multi-Strategy1	83.02	83.02	85.85	94.34	70.75	56.60	—-	90.57	94.34
Steelix	67.72	70.08	70.87	74.80	61.42	47.24	75.59	—-	97.64
Multi-Strategy2	65.91	66.67	68.18	75.00	59.09	45.45	75.76	93.94	—-

5.5.1 Dataset

To explore our research questions, we chose a varied dataset that is amenable to large-scale experiments. Specifically, we use the dataset of vulnerable programs produced for the DARPA Cyber Grand Challenge (CGC) [82] for our experimentation. These binaries contain a very diverse set of functionalities and variable complexity, and guarantee the presence of known bugs [83, 84], providing a perfect testing ground for our system. After filtering out challenges that involved multiple binaries (which AFL cannot currently fuzz) and challenges which failed to build for Linux (using a port of the CGC dataset to that platform [85]), there were 217 different binaries that we used.

To evaluate the impact of abstraction functions on *real-world* software, we evaluated them on Objdump and ImageMagick in Section 5.5.5.

Table 5.3: Results of each of fuzzing using the different abstraction functions on the CGC dataset.

Selection Strategy	Type	Number Crashes	Mean Block Coverage	Mean Execs per sec
Basic Blocks	SAF	91	38.1%	625
Edges	SAF	91	38.5%	620
Block Triples	SAF	92	39.0%	412
Edge + Return Loc	SAF	101	39.5%	555
Function Context	SAF	78	35.5%	569
Method Calls	SAF	61	28.6%	469
Multi-Strategy1	MAF	106	39.5%	551
Steelix	MAF	127	47.2%	581
Multi-Strategy2	MAF	132	48.1%	530

5.5.2 Experimental Setup

We evaluated each binary in the CGC dataset using each of the seven selection strategies detailed in Section 5.3). Each of these single-strategy fuzzing *configurations* was run for 8 hours on 12 cores, giving each configuration 96 CPU hours of experimentation time. This is well over the minimum time suggested by [86].

We also evaluated two *multi-abstraction fuzzing* (MAF) configurations that combines multiple select abstraction functions. In each of these, the abstractions were run in parallel (with cross-fuzzer synchronization of $\widetilde{I}$) for 8 hours, with 12 cores total divided equally between the fuzzers. Note that this is the same amount of resources provided for testing the single-abstraction fuzzers. The abstractions used in the MAF configurations are shown in Table 5.1. We chose one configuration (*Multi-Strategy1*) which included every abstraction function except *Steelix*, and one MAF configuration with *Steelix* (*Multi-Strategy2*). The abstraction functions chosen for *Multi-Strategy2* aimed to pick the a smaller, but varied set of selection strategies.

5.5.3 Crash Numbers

Table 5.3 shows the total number of binaries crashed by each SAF configuration, along with the MAF configurations. Confirming our intuition, both multi-strategy fuzzers performed better than any of the single-strategy fuzzers that were part of them. The three best fuzzers (*Multi-Strategy2*, *Steelix*, *Multi-Strategy1*) were all MAF's, indicating that MAF fuzzers perform better than their SAF counterparts. The *Steelix* strategy performed very well, crashing a total of 127 binaries. This result was expected because it is designed to recover strings and magic numbers, of which there are many in the CGC. The *Multi-Strategy2* strategy performed the best, crashing 132 of the 217 binaries. The one SAF that performed the best was *Edge + Return Loc*, crashing 102 binaries, 10 more than any other SAF. This result shows that adding some callstack context to the abstraction seems to improve it's bug-finding capabilities.

The *Method Calls* configuration was the least effective with only 55 crashes. As explained in Section 5.3, this abstraction function is specialized for object-oriented programs, while most of the CGC binaries are not object-oriented.

These results allows us to infer an answer to Research Question 1 (RQ1).

> **Answer for RQ1**: The choice of an abstraction function is important in fuzzing, and an inappropriate abstraction function (e.g., *Method Calls* on non-object-oriented code) can seriously impair fuzzing effectiveness.

The relatively poor performance of the *Method Calls* and *Function Context* SAF configurations does not mean that these abstractions are useless. In fact, we found in our evaluation, when combined with other abstraction functions in *Multi-Strategy1*, it enabled the detection of a crash in Modern_Family_Tree which no other configuration found.

Table 5.2 shows a fine-grained comparison of different configurations. It contains the percentage of binaries that are crashed by the configuration in the corresponding row that are *also* crashed when using the configuration in the corresponding column. Specifically, the value of

each cell at row m and column n is computed as: $\frac{C_{AF_m} \cap C_{AF_n}}{C_{AF_m}} * 100$, where C_k is the total binaries crashed when using the abstraction function k, AF_m, and AF_n are the abstraction functions of row m and column n respectively. This table allows the comparison of the different abstraction functions against each other. For a given pair of abstraction functions α_m and α_n), we can make following observations based on the value of a cell (m, n), which is the value at row m and column n:

- A higher value at the cell (α_m, α_n) indicates that α_n encompasses the effectiveness of α_m.
- A higher value at the cell (α_n, α_m) indicates that α_m encompasses the effectiveness of α_n.
- A higher value at both the cells (α_m, α_n) and (α_n, α_m) indicates that the abstraction functions α_m and α_n have similar bug finding abilities.
- A lower value at both the cells (α_m, α_n) and (α_n, α_m) indicate that the abstraction functions find different bug types and these are the good candidates to be combined.

The *Multi-Strategy1* configuration was able to leverage the capabilities of the six abstraction functions that it utilized, and this can be seen from the high percentages in the cells of the *Multi-Strategy1* column and relatively low percentages in the cells of the *Multi-Strategy1* row. This is true despite the fact that the individual abstraction functions in *Multi-Strategy1* receive only a fraction of the time they had individually.

If we look at the *Steelix* column we see that the only SAF's that were not very highly encompassed by it were *Edge + Return Loc* and *Basic Blocks*. *Basic Blocks* being a more coarse-grained abstraction might've done better in cases where *Steelix* produced too many paths to process. This result also implies that combining *Steelix*, *Basic Blocks*, *Edge + Return Loc*, *Function Context* in the *Multi-Strategy2* configuration is a good choice.

Looking specifically at the column for *Multi-Strategy2*, we see that it encompassed all the other strategies fairly well; *Multi-Strategy1* was the only strategy that it didn't find at least 95% of the same crashes. This implies that although we were able to capture most of the bug finding capabilities of the other strategies with this combination, we still missed some bugs that a different combination of strategies got.

Answer for RQ2: The results show that both *Multi-Strategy1* and *Multi-Strategy2* configurations were more effective at finding crashes than their individual abstraction functions. Therefore, combining abstraction functions, is an effective technique to enhance fuzzing.

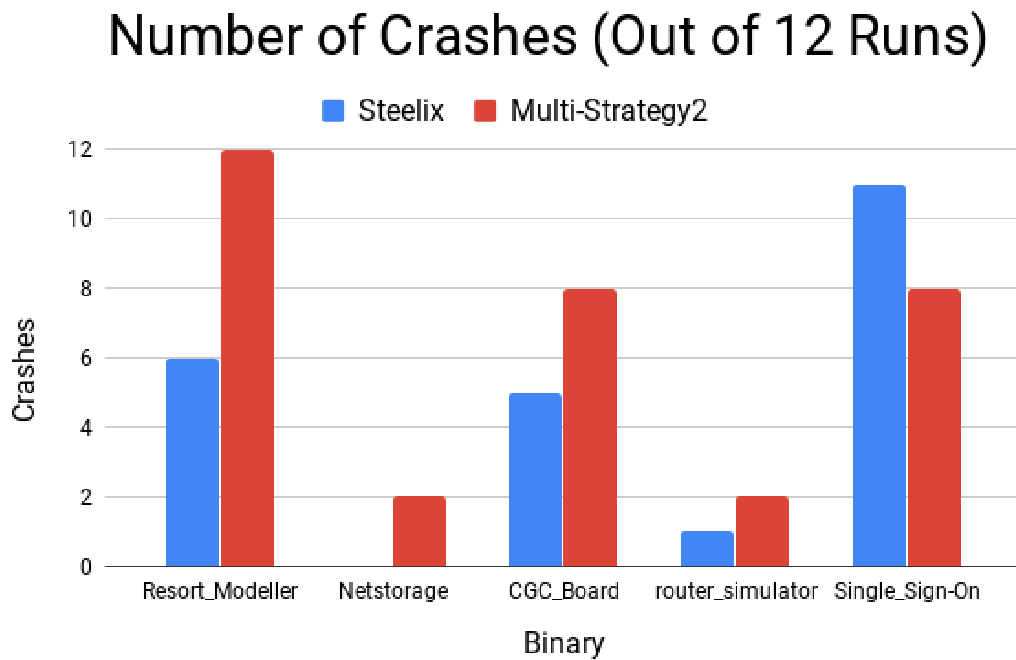
Abstraction functions based on basic blocks (*Basic Blocks*, *Edges*, *Block Triples*, and *Edge + Return Loc*) all show similar bug finding abilities, and this is evident from the high percentage in the cells of the corresponding abstractions. Finally, it is interesting to see that having more fine-grained abstractions (e.g. block triples, vs edges) does not necessarily improve the effectiveness.

5.5.4 Different Strategies Repeatedly Crash Different Binaries

In the evaluation, we saw plenty of cases in which a binary is crashed by one configuration and not another. This is seen in Table 5.2, anywhere where the overlap is not 100%. We want to evaluate whether these results are random or if binaries exist where one configuration is for sure better at finding that crash.

First we explore *Multi-Strategy2* vs *Steelix*, to know if the binaries crashed by the first and not the second are random or repeatable. To test this, we picked five binaries that were found by *Multi-Strategy2* and not *Steelix* and explored what happened if we repeated the test 12 additional times. These results are shown in Figure 5.1. Note that the original fuzzing run is not included to make sure the results aren't biased. We see that of the five binaries that in only one case did *Steelix* do better in our repeated evaluation. This result shows that *Multi-Strategy2*

Figure 5.1: This graph shows the results of re-running the *Steelix* and *Multi-Strategy2* fuzzer configurations 12 times on five select binaries that were only found by *Multi-Strategy2*. These results do not include the original large-scale test.



outperforming it on these binaries is repeatable.

Even when comparing individual abstraction functions we were frequently able to find examples where one abstraction function repeatedly did better than others. Due to cost and time constraints we weren't able to re-run all tests twelve times like above, but here's some examples we tested.

- Dive_Logger was crashed at least 10/12 times by *Edge* + *Return Loc* and each of the MAF's that include *Edge* + *Return Loc*, but rarely by other strategies.
- Modern_Family_Tree was crashed 11/12 times by *Multi-Strategy1*, but no more than 3/12 times by any other strategy (including *Multi-Strategy2*).
- Neural_House was rarely crashed by *Edges* and *Basic Blocks*, but frequently by *Block Triples*, *Edge* + *Return Loc* and *Steelix*.

These results that sometimes a fuzzer which in general doesn't find the most crashes might be better at a specific binary. It also indicates that the incomplete overlap we saw in Table 5.2 is a product of the different performance of the fuzzing configurations.

5.5.5 Real World Programs

We evaluated the effectiveness of the individual abstraction functions, as well as the effectiveness of the multi-abstraction fuzzers on two real-world programs: Objdump-2.26.1, and ImageMagick-7.0.4-2. These are programs that have been used in other fuzzing evaluations [86].

We ran each fuzzer configuration described previously six times on each of these programs. Each run was using 12 cores for 8 hours for a total of 96 core-hours. Then we analyze the results in terms of unique number of crashes, where unique is defined in terms of the crashing

Table 5.4: Results of fuzzing using different abstraction functions on ImageMagick and Objdump. Note that there were no crashes on ImageMagick without seeds so those columns are omitted. Results shown are the median of six runs.

	ImageMagick 7.0.4-2 (seeds)	Objdump 2.26.1 (seeds)	Objdump 2.26.1 (no seeds)
Selection Strategy	Median crashes	Median crashes	Median crashes
Basic Blocks	0.5	**7.0**	3.0
Edges	0.0	**7.0**	4.0
Block Triples	1.0	5.5	**4.5**
Edge + Return Loc	1.5	6.5	4.0
Function Context	1.0	4.0	1.5
Method Calls	0.0	3.0	1.0
Multi-Strategy1	1.0	5.0	2.5
Steelix	1.0	6.0	3.5
Multi-Strategy2	**2.0**	6.0	**4.5**

callstack. Note that these do not represent unique bugs, as attributing crashes to bugs is out of scope of this paper. These programs were fuzzed both with and without seeds.

The results are shown in Table 5.4. As we can see from the table, the strategy that does the best depends both on the binary being fuzzed and whether or not seeds are provided. *Basic Blocks* and *Edges* do better than any other configuration on Objdump with seeds, however, they both are significantly less effective without seeds and perform very poorly on ImageMagick. Looking into the results, it seems that for Objdump, especially with seeds, these two strategies had generated significantly less inputs compared to say *Steelix*, and executed inputs about 20% faster. Since the extra capabilities of *Steelix* and *Edge + Return Loc* weren't necessary to trigger additional coverage (especially when given seeds!) the faster basic sensitivities were the best.

One consistent trend across all of them is that *Multi-Strategy2* does very well, even if it is not always the best. This confirms our earlier results that it is widely applicable on a range of binaries. *Multi-Strategy2* has the abilities of *Basic Blocks* to explore lots of inputs quickly and

that of *Steelix* and *Edge + Return Loc* to deeply explore. Another interesting point is that *Edge + Return Loc* did well on all four tests. This implies that some callstack context is important for fuzzers to be able to explore lots of the state-space.

5.6 Discussion

We saw in our evaluation of fuzzing strategies that combining different abstraction functions allows them to complement each other by sharing inputs between them. This resulted in the multi-abstraction fuzzers crashing more binaries than any single strategy from their components. Even though the *Multi-Strategy2* configuration found the most crashes in our tests, there were programs in which another configuration was more likely to crash. Future work could be to try and automatically determine the best fuzzer configuration to use. Also, as mentioned in the framing of the evaluation, other input generation techniques, such as Driller, could also be used on top of this to improve results for all abstractions in our evaluation.

Despite finding that *Multi-Strategy2* found the most crashes in our tests, we also saw programs in which a different configuration was more likely to find a crash, as was discussed in Section 5.5.4. One takeaway of this is that although a particular fuzzer might find less crashes than another it still might be useful if it finds *different* crashes. Consider *Edge + Return Loc*, alone it finds less crashes in number than *Steelix* but it also found crashes in *different* programs.

5.7 Conclusion

In this paper, we have presented the first formalization of input evaluation in fuzzing, borrowing concepts from the field of static analysis. This formalization can immediately be used as an effective base for future research: by reasoning about which concepts of the formalization have been explored by current work (and the more salient question of which have not been), we

identified that the impact of different *abstraction functions* on fuzzing outcomes is unexplored in current work.

Thus, we performed an investigation into alternate abstraction functions for fuzzers. We identified seven input abstraction functions with various levels of granularity and evaluated them on a large dataset from the DARPA Cyber Grand Challenge and on two real-world programs. The results show that the choice of an abstraction function is important and can affect the effectiveness of fuzzing. Furthermore, we show that combining different abstraction functions is superior to using just one.

Keeping with the scientific spirit, we open-source the resulting abstraction-modular fuzzer.

Chapter 6

Token-Level Fuzzing

6.1 Introduction

As the amount of software in the world grows, so does the need for effective automated bug-finding techniques. It's incredibly common for companies to employ far more developers than security engineers. BSIMM, a study of software security initiatives started by Synopsys, found that there was an average ratio of a single security engineer for every 60 software developers [87]. Consequently, security engineers are often responsible for very large amounts of code; far more than is feasible to check manually. As a result, it is imperative that effective automated techniques are used to uncover many of these bugs.

In the past few years, fuzz testing has become widely popular. Fuzzers such as American Fuzzy Lop [29], Syzkaller [88], and Libfuzzer [89] are responsible for the detection of hundreds of high severity security issues. The success of these fuzzers as well as others, has caused fuzzing to become the preeminent automated analysis for detecting memory corruption vulnerabilities. It is employed by companies for both finding old bugs and as continuous integration checking for newly introduced bugs [90].

Similarly, fuzzing research has steadily taken off as people have built a myriad of fuzzers

for a variety of targets. One target of interest in particular is interpreters. Interpreters are in widespread use; they are found in many components of browsers, pdf readers, document viewers, programming languages and more. As such, these are often a high value target for attackers, and a high impact target for security researchers.

Modern-day interpreters can be hugely complex, for example, V8, Google's JavaScript engine, is over 700K lines of code! The huge amount of code involved and the complexity of the engines themselves, make them difficult to manually audit, and consequently, a very attractive target for fuzzing. However, interpreters present some unique challenges when it comes to fuzzing.

Interpreters can be different from other types of programs in that they expect highly structured inputs made up of individual *tokens*. If the input does not match the syntax that the interpreter is expecting, then the input may throw an error and the interpreter will not run the code any further. As such, many of the most common fuzzers fail to perform well when applied to interpreters, such as JavaScript engines. Their mutations typically result in simple syntax errors and they fail to make much progress in mutating the input seeds. Even generating an input that parses successfully is rare.

Because of the aforementioned issue, most fuzzers are specifically targeted to the task, and the most common approach for this is to use grammar-based fuzzers [91–94]. Grammar-based fuzzers require information on the grammar that the interpreter expects, and then by following the grammar, in generation and mutation of inputs, they can create test cases which exercise deeper code paths in the interpreter. These approaches are effective but suffer from some limitations. For one, they need to be given or be able to learn a grammar, which makes it difficult to re-target for a different language.

Another limitation with grammar-based fuzzers is that they frequently conform too tightly to the supplied grammar and fail to generate unusual situations for the parser. To expand on that, grammar-based fuzzers either are given or learn a grammar. That grammar is used to

generate inputs. However, if a bug is caused by an input that does not conform to the grammar it may be difficult or impossible for the grammar-based fuzzer to trigger it.

In this paper, we introduce a novel technique, called *Token-Level Fuzzing*. Token-Level Fuzzing can be thought of as a level in between the byte-level approaches, and the grammar-based approaches typically employed for fuzzers. The basic idea behind it is to have the mutations work with whole tokens, either replacing or inserting entire words. For example instead of replacing a couple random bytes, which has little chance of producing an interesting input, we could replace a couple tokens in the input with different tokens. This idea allows the fuzzer to have a much higher chance of producing useful mutations, which are more likely to trigger new interesting testcases, while avoiding the strictness and complexity of grammar-based approaches.

We created a modified version of AFL, called *Token-Level AFL*, which implements this new technique. Token-Level AFL is specifically implemented for fuzzing JavaScript engines, although the technique itself is general. We test it against the most up-to-date versions of the four major JavaScript engines: V8, SpiderMonkey, JavaScriptCore, and ChakraCore. In doing so, we find 27 bugs across the engines, many of which are severe and can lead to remote code execution.

In summary, this paper makes the following contributions:

- Introduces a new technique, called Token-Level Fuzzing, for fuzzing language-based programs, such as interpreters.

- Implement this technique to fuzz JavaScript engines. The implementation is done on top of AFL to take advantage of its efficient coverage guided fuzzing.

- Evaluate Token-Level AFL on the latest versions of the four major JavaScript engines, finding 27 previously unknown bugs.

- Compare the fuzzing results against other state of the art JavaScript fuzzers.

6.2 Motivation

6.2.1 Ineffectiveness of Byte-Level Fuzzing

As discussed in the previous section, fuzzing research has come quite a long way from just generating purely random input. AFL in particular is a venerable fuzzer with many bugs found in over one hundred highly used targets. However, when AFL is applied to interpreters, such as JavaScript engines, some significant downsides begin to emerge. As most of the mutations that AFL performs are at a byte or bit level, we see it repeatedly generating inputs that simply fail to parse, often simply because the input now contains incoherent tokens.

If we consider a simple bitflip mutation on a small piece of JavaScript, the results will frequently look like the following mutations, which will immediately fail to parse:

```
1 while (bar.x) ⟶ whkle (bar.x)
2               ⟶ whilep(bar.x)
3               ⟶ while xbar.x)
4               ⟶ while (bar.|)
```

It should be straightforward to see that mutations such as this are not particularly helpful; they will only trigger simple error handling. As such, this mutation would very likely not lead to more code coverage, and would simply be wasted execution time. This is not an uncommon issue, we can expect most mutations performed by AFL to result in simple syntax errors, and only a tiny fraction of the mutations will actually trigger new code coverage. Thus, AFL will waste the majority of its execution time on mutations such as these.

6.2.2 Grammar-Based Fuzzing

Given the highly-structured input required for JavaScript engines (and interpreters in general), most work uses a grammar-based approach. Grammar-based fuzzers are incredibly powerful in their ability to very quickly generate syntactically correct pieces of input for a given program. Given a grammar definition, these fuzzers use that definition to generate inputs which will be executed by the target program. An obvious downside with this approach however, is the work required to first define a grammar, or otherwise rely on an existing grammar definition before fuzzing can be performed [92, 95–98].

```
function main() {
  const v1 = [13.37,13.37,13.37];
  const v6 = [1337,1337,v1];
  function v9(v10,...v1) {
    const v13 = [1337,1337,1337];
    return v13;
  }
  const v18 = v9(v6);
}
main();
```

Figure 6.1: listing
Example of code generated by fuzzilli. Fuzzilli follows a static single assignment format for the generated code. As such, variables will always be assigned to exactly once and some syntactic/semantic patterns cannot be emitted.

An additional downside to grammar-based fuzzing is the adherence to the grammar that was giving to the fuzzer. This not only limits the fuzzer to creating code that matches the grammar, but it *also* limits the fuzzer to finding bugs that can be written as such. This will prevent most grammar based fuzzers from finding bugs that *require* syntactically or semantically incorrect input to trigger. Even bugs with unusual semantics can be unreachable by grammar-based fuzzers. This is because a grammar-based fuzzer, though powerful in its generational capabilities and language awareness, will generate inputs that adhere to the grammar that has been supplied.

To explain that further we will show an example from Fuzzilli and talk about how its grammar limits the bugs it can find. Listing 6.1, shows an example input generated by Fuzzilli, which was taken when fuzzing a JavaScript engine. Note how each line assigns at most a single new variable and variables are never overwritten. This is because Fuzzilli uses a static single assignment intermediate representation [99], and the inputs it generates will conform tightly to it. This feature both enables the real-world results that Fuzzilli has published, but it also limits the sorts of bugs that it is able to find. Any bug that requires a different or more complicated structure such as redefining variables, will not be generated. Furthermore, Fuzzilli will never create nested expressions and cannot output many of the syntax errors that can be found in JavaScript.

6.2.3 Bugs Requiring Incorrect Semantics

Unsurprisingly, there are bugs which do require incorrect semantics or even incorrect syntax, as well as bugs that require unusual constructs. We will briefly look at an example of such a bug which was found in V8. Chromium issue 800032 [100] describes a high impact bug found in v8, which could lead to remote code execution. Note that although the bug has high impact with potential for RCE, no CVE was assigned as it was discovered internally by Google Project Zero member Jung Hoon Lee. The bug report includes the proof of concept in Listing 6.2, which triggers the issue.

The proof of concept creates a subclass of a Regular Expression object, and in the constructor of the subclass there is an error. The line, `const a = 1`, will attempt to redefine `a` as constant, which is not correct semantics. Because of this syntax error, the size of an object gets incorrectly computed which can then lead to out of bounds reads and writes on the object. Sticking to a strict grammar will prevent us from finding issues such as this.

Another example of a bug that could be difficult to find with a grammar-based fuzzer which

```
class Sub extends RegExp {
  constructor(a) {
    // expected_nof_properties() skipped
    // due to error
    const a = 1; // semantic error
  }
}

let o = Reflect.construct(RegExp,[],Sub);
// OOB write
o.lastIndex = 0x1234;
```

Figure 6.2: listing
Proof of concept code for Chromium Issue 800032. This code triggers a semantic error, which causes a miscalculation in the number of properties leading to an exploitable out-of-bounds write.

adheres too tightly to a grammar is shown in Listing 6.3. This example is CVE-2017-8729 of Edge [101], where the parser would incorrectly parse the code, and in doing so, lead to a type confusion when assigning to the object member later. As this bug requires incorrect syntax to trigger, this example showcases another case in which grammar-based fuzzers may suffer due to their adherence to the grammar.

6.2.4 A Middle Ground

We have just shown how grammar-based fuzzers may be unable to find certain bugs in interpreters, and previously, in Section 6.2.1, we showed how Byte-Level fuzzers, such as AFL, struggle to make any progress in fuzzing language-based inputs. It is apparent there is a need for something in the middle, that can make progress and explore interpreters effectively, but without the limitations of a grammar. In order to find a way to utilize the powerful evolutionary capabilities of tools like AFL on language-based inputs, we introduce a new technique, *Token-Level Fuzzing*. Token-Level Fuzzing works on a higher level than bytes, but not at a full grammar level, allowing it to find bugs neither technique would find.

```
function f() {
  ({
    a: {
      b = 0x1111, // invalid assignment
      c = 0x2222,
    }.c = 0x3333
  } = {});
}

f();
```

Figure 6.3: listing
Proof of concept code for CVE-2017-8729, which was caused by a parser error in Edge. Line 4 (`b = 0x1111`) contains a syntax error by trying to assign to a member with = while creating an object.

6.3 Overview

6.3.1 Token Level Fuzzing

The idea behind Token-Level Fuzzing is fairly simple: Valid tokens should be replaced with valid tokens. So when fuzzing the example given in Section 6.2 instead of mutating individual characters in the word `while`, we would want to replace the entire word with a different word. For example if we replaced it with `if` or `Number`, it would be a much better mutation. Here's an example of possible better mutations if we use Token-Level Fuzzing:

```
1 while (bar.x) ⟶ if (bar.x)
2               ⟶ Number (bar.x)
3               ⟶ while (bar+x)
4               ⟶ while (while.x)
```

Notice that Token-Level Fuzzing can still produce invalid syntax such as the last one above which has `while (while .x)`. Even mutations like that can be beneficial if they trigger a new error handler or if they can iteratively be mutated until a different valid JavaScript statement is reached.

Comparison to dictionaries

A natural question is to ask how does this technique compare to the "dictionary" that tools such as AFL [102] and LibFuzzer [30] allow users to provide. The first major difference is that AFL will still do the Byte-Level mutations as well as the dictionary based mutations. Secondly, the dictionary mutations are not aligned to tokens, so it might insert the word `while` in the middle `function` instead of replacing the whole token. Finally, it may take multiple token additions/replacements to reach a new interesting input, some fuzzers such as AFL, will only insert one dictionary word in a mutation, limiting its exploration.

Comparison to grammar-based fuzzing

Grammar-based fuzzing mutates inputs or generates inputs according to a grammar, whereas Token-Level Fuzzing does not follow any grammar. Token-Level Fuzzing can generate many patterns that can be difficult or impossible for a particular grammar-based fuzzer, in particular those with complex or incorrect syntax. On the other hand, grammar-based fuzzers focus on exercising the interpreter with correct syntax, possibly allowing faster exploration of that part. As a result, we expect that our technique will find different bugs.

6.3.2 Method

To create a fuzzer which works on a *Token-Level*, we start by constructing a map, which assigns each possible token in the language a unique numerical value. Then we can *encode* input files into a list of numbers, which are the encoded version of the seeds. Fuzzing is then performed on this list of numbers, and changing any number to a different number is equivalent to replacing it with a different token. Whenever we want to run against the target (JavaScript engine) we need to transform the mutated list of numbers back into the original language. This is done with a *decode* function which replaces each number with the corresponding token and

Figure 6.4: The architecture of Token-Level AFL. It is made up of two primary components: The preparser and the fuzzing engine. The preparser is responsible for transforming input seeds into a list of 16 bit numbers. Then the fuzzing engine works on these lists, only decoding them back to JavaScript to execute.

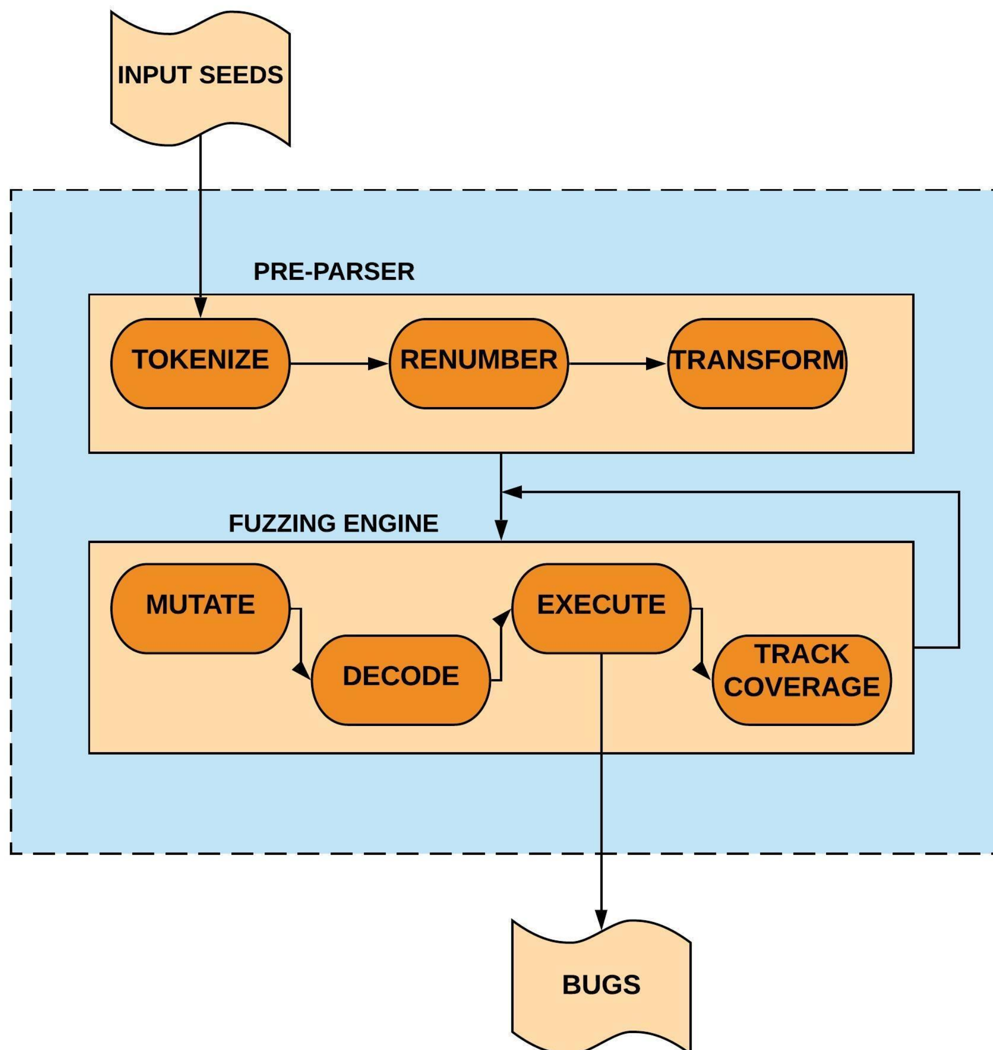
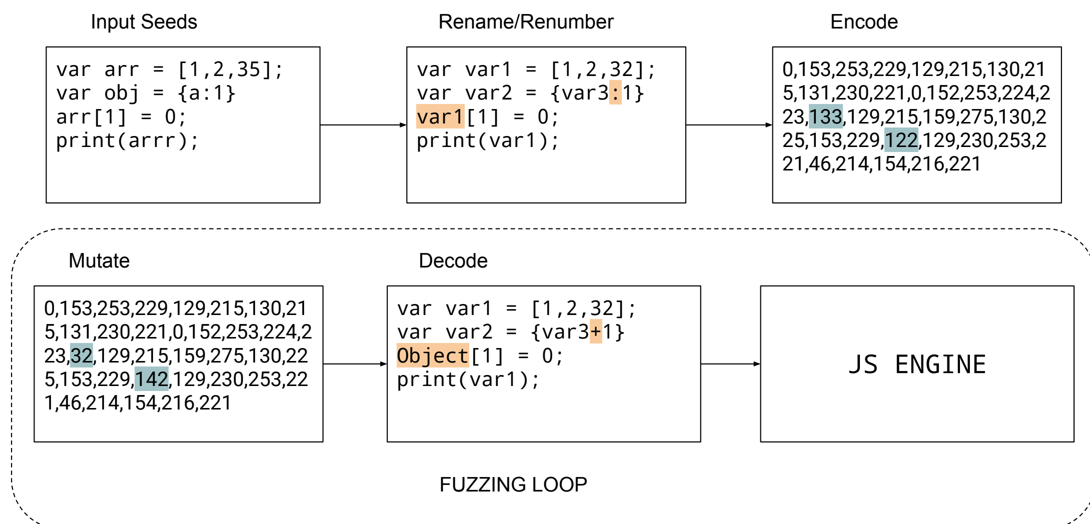


Figure 6.5: An example of what happens to a single seed in Token-Level Fuzzing. The seed first goes through the renaming and encoding stages which produce a list of numbers. Then when running in the fuzzing loop, it is mutated and decoded prior to execution, where coverage feedback will determine if the input is added to the queue or mutated further. We highlighted how changing a couple numbers in the encoded form results in completely different tokens in the decoded result.



concatenates them with spaces as needed. Thus, fuzzing can be done on the list of numbers without any knowledge of what they mean.

Of course we need to consider one last thing, that is the list of valid tokens is infinite for many languages as it includes: all possible numbers and all possible variable names that are legal in the language. If the token-map contained too many numbers than it would unnecessarily slow down the fuzzer, because most tokens would be numbers and only very few would be other functionality. To remedy this, we pick a small set of valid numbers consisting of all powers of two, as well as numbers plus/minus one of a power of two. Similar values have been chosen for other fuzzers, such as Difuze and AFL, to reduce the number of possible inputs [29,77]. Similarly, we found by looking through regression cases that only a small number of variables were needed to trigger most bugs, so we limited the number of variable names to fifteen possible names.

6.3.3 Implementation

Our implementation of Token-Level Fuzzing is done on top of AFL, to take advantage of the coverage guided nature of AFL. The resulting tool is called *Token-Level AFL*. Token-Level AFL is the combination of two components: a preprocessor written in python which analyzes the tokens of the input files and encodes them for fuzzing, and a modified version of AFL which performs fuzzing on the encoded inputs. Figure 6.4 shows the overall architecture of this as well as its main components.

The preprocessing step is written in python and runs the following steps:

Rename For each input seed, variable names are randomly replaced with one of the fifteen pre-defined variable names: var1, var2, ..., var15. Variable names are not repeated unless all fifteen variable names have been used already.

Renumber As described earlier, we limited the set of valid numbers to a pre-defined set. All

numbers are replaced with the closest number from that set.

Token Analysis Use a JavaScript Lexer to find all the tokens used in all the seeds. Assign each token a numerical value, which will be its encoded value.

Encoding Transform each input into a list of numbers by replacing each token with its encoded value. This list is then flattened by encoding each value as a 16 bit integer.

After the preprocessing step to generate the token mapping and the encoded seeds, fuzzing occurs on the encoded inputs. All of the mutations occur on the encoded inputs, which allows us to keep most of the fuzzer code unchanged for the Token-Level modifications. This step is done with minor modifications to AFL:

Mutations Mutations are slightly modified to work on an array of 16 bit numbers rather than an array of bytes. 16 bit numbers were necessary because there were more than 256 tokens. Note that this change is very small; it is effectively just changing the type of the array to `short*` rather than `byte*`.

Decoding The input is decoded immediately before executing the input in the target JavaScript engine. This small shim simply concatenates the tokens together, adding spaces as needed[1].

Figure 6.5, shows an example of the various steps of Token-Level AFL when applied on an input file. The preprocessing steps are used to change the inputs into their encoded forms. From there the mutations happen entirely on the encoded inputs. The encoded inputs are only decoded when run in the target program.

[1]No spaces are added for certain tokens such as quotation marks

6.3.4 Further Mutation Modifications

Some of the mutations that AFL performs are not very useful or applicable to fuzzing on the intermediate list in Token-Level Fuzzing. Some examples of these are are the arithmetics and interesting number strategies. In these strategies AFL will try inserting interesting numbers such as ”1024”, ”2147483647”, ”-100663046”, etc into the stream of bytes. Because these will get translated to a series of tokens, this just will add a constant random list of tokens into the fuzzed input. As such, we remove these strategies which do not apply well to our scenario.

Of course, then the obvious question is are there strategies we should add that do apply better to Token-Level Fuzzing. One simple strategy that we identified is to randomly insert and overwrite multiple tokens in a row. The intuition behind this is that changing one token at a time may not be enough, it may be necessary to change more than one to create a new interesting input. Because this was not happening sufficiently with AFL's current set of fuzzing strategies, we added the following mutation strategies to the fuzzer:

Random Insert Randomly inserts up to three new tokens somewhere into the file being mutated.

Random Overwrite Randomly overwrite up to three tokens in a row in the file with the same number of new tokens.

Random Replace Randomly replace up to three tokens in the file with up to three new tokens. Note that this strategy can insert more tokens than were removed.

6.4 Evaluation

To evaluate our implementation of *Token-Level Fuzzing*, we run the fuzzer on JavaScript engines from the four major browsers, V8, SpiderMonkey, JavaScriptCore, and ChakraCore[2].

[2]ChakraCore is no longer used in Edge as of January 2020 [103]

Our goal is to understand its bug-finding capabilities as well as how our implementation compares to other state-of-the-art JavaScript engine fuzzers. In order to reason about these goals, we formulate the following research questions which we will answer:

RQ1: Does Token-Level Fuzzing generate more syntactically correct inputs than Byte-Level Fuzzing?

RQ2: How does Token-Level Fuzzing compare to other state-of-the-art Fuzzers?

RQ3: Is Token-Level Fuzzing able to find real-world vulnerabilities in the latest JavaScript Engines.

RQ4: Do bugs found by Token-Level Fuzzing involve incorrect syntax/semantics?

6.4.1 Experiment Setup

We downloaded the latest available versions of the four major JavaScript engines as of October 1, 2019. These were the development versions cloned from the official git repositories. We compiled all engines with debug checks. Debug checks are additional checks that the programmers include to try to catch unexpected conditions [104], so we enabled them for fuzzing to catch more potential security bugs. We did not enable Address Sanitizer or other santizers as these tended to be too slow in our tests.

Seed Collection Having good seeds is essential for our fuzzer for multiple reasons. Firstly, the list of potential tokens which will be used by our fuzzer are gleaned from the input files. Thus, it is essential that the seeds cover as many of the tokens used by the language as possible. Secondly, our implementation of Token-Level Fuzzing is based on AFL and evolutionary fuzzing, so having a quality set of diverse seeds helps the fuzzer greatly, because it will explore starting from these initial seeds. To collect seeds, we manually selected regression tests from

the repositories of the various JavaScript engines. We picked seeds covering a wide range of functionality, but limited the number of seeds to 100 total seeds.

Comparison with other tools. We compared Token-Level AFL against the following state-of-the-art tools: AFl [29], Fuzzilli [99], and CodeAlchemist [95]. For each tool, we ran it for three days on 30 cores, on each of the four major JavaScript engines, resulting in a total of 2160 core-hours for each fuzzing run. Each Fuzzer-JavaScript engine combination was run three times to limit randomness in our experiments. Note that Fuzzilli does not provide a mechanism for using seeds, so it was run without seeds. On the other hand, CodeAlchemist used far more seeds in their paper [95], so we did a much larger automated seed collection, grabbing all javascript files from regression tests, resulting in 32682 seeds.

6.4.2 Syntactically valid inputs

The most basic assumption of Token-Level Fuzzing is that it generates more inputs which are syntactically correct than Byte-Level Fuzzing and that these inputs will, in turn, trigger deeper functionality. To understand the validity of this assumption we first compare the results of AFL and Token-Level AFL. Both fuzzers were given the same seeds and AFL was given all of the tokens in the input files as a dictionary. With a dictionary, AFL will try inserting the keywords in the mutation steps. This allows AFL to make some progress on languages such as JavaScript, and showcases the best configuration for AFL [102]. In our experiments, even with a full dictionary and the same input seeds, AFL was only able to find 2 bugs across all the JavaScript engines, whereas Token-Level AFL found 14!

We added tracking to determine how many of the generated inputs of each fuzzer parse successfully or hit a parser error. These numbers are shown in Table 6.1. 10.68% of all runs of AFL resulted in parser errors! This shows that as we suspected in Section 6.2, most inputs generated by AFL fail to parse and trigger any reasonable functionality in the JavaScript

Table 6.1: This table shows what fraction of inputs generated by AFL and by Token-Level AFL are able to be parsed successfully when fuzzing V8. The higher parse rate of Token-Level AFL shows that by mutating tokens instead of bytes, our technique is able to generate more correct inputs.

Fuzzer	**Successful Parses Rate**
AFL	10.68%
Token-Level AFL	30.26%

Table 6.2: Average number of bugs found by each of the tested fuzzers on the four major JavaScript engines. (Fuzzilli does not have code for running on ChakraCore, so that table entry is omitted).

	V8	JSC	Spider-Monkey	Chakra-Core
Token-Level AFL	4.67	0.66	0.66	2
AFL	0	0	1	0.33
CodeAlchemist	0.66	0	0	3.33
Fuzzilli	0	1.33	0	N/A

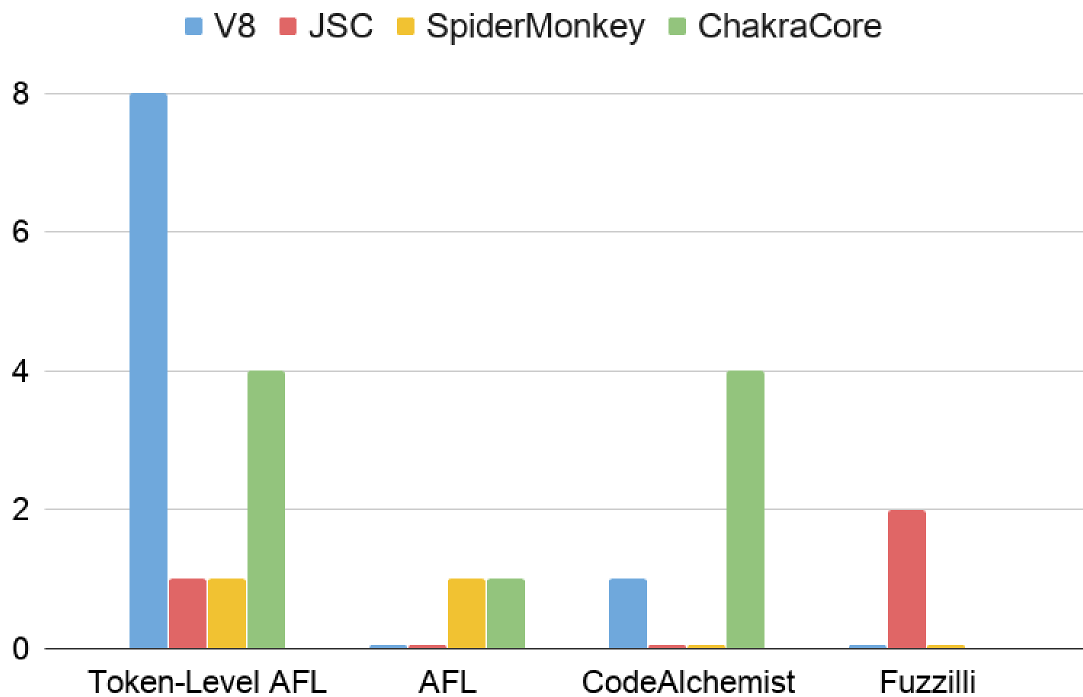
Engines. The difference provided by Token-Level Fuzzing is immediately evident; 30.26% of all inputs generated by Token-Level AFL were successfully parsed. The higher fraction of successfully parsed inputs by using Token-Level Fuzzing shows that it allows the fuzzer to generate far many more inputs that trigger useful functionality. This in turn should let the fuzzer find deeper bugs and explore more of the javascript interpreter.

Answer for RQ1: The results show that Token-Level Fuzzing generates syntactically correct inputs about three times more often than Byte-Level Fuzzing, enabling much more efficient fuzzing of interpreters.

6.4.3 Comparison with other State-of-the-Art Fuzzers

In this section we will explore how Token-Level AFL performs when compared against other state-of-the-art JavaScript Engine fuzzers. For this comparison, we selected AFL, Fuzzilli [99], which is maintained by Google's Project Zero, and CodeAlchemist [95], a recent paper on

Figure 6.6: This graph shows the total number of unique bugs found by each of the tested fuzzers when run on the four major JavaScript engines for a time period of 72 hours. This is an aggregate number of bugs across the three runs, and only unique bugs are counted.



JS engine fuzzing. We evaluated all of these fuzzers on the latest available JavaScript engines which were retrieved from the official repositories on Oct 1, 2019. Each test was run on 30 cores for 3 days.

As is usual for fuzzing research, we will use the number of bugs found as the main performance metric. We consider any debug check, release check, or memory corruption to be a bug for our purposes. Although debug checks may not always indicate a security issue was found, they do indicate that an assumption was violated, and they show that a fuzzer is finding bugs which have not been previously found and fixed by the vendor. To identify unique bugs we filter based on any asserts hit, as well as manual analysis to ensure only unique issues are counted.

Additionally, we investigate block coverage during this evaluation. Although block coverage may not be as meaningful a measurement as number of bugs found, it still shows useful information [86]. To be able to trigger a bug, a fuzzer must be able to trigger the code where the bug lies. So coverage is a necessary but not sufficient condition for finding bugs and can be used as a performance metric. We collected block coverage information throughout each of the fuzzing runs with minor modifications and post-processing after the fuzzing run.

Results: As shown in Figure 6.6, Token-Level AFL found the most crashes during the 3-day fuzzing periods. Token-Level AFL found 14 total bugs across the three runs, while the second best performer, CodeAlchemist, found 5 bugs. Additionally, only one of the 14 bugs found by Token-Level AFL was found by another tool! Each of the other 13 bugs were unique to it. So although CodeAlchemist also found four bugs in ChakraCore, none of those bugs overlapped with the four found by Token-Level AFL. This indicates that our method finds bugs which other fuzzers are not currently able to find.

When investigating coverage we found that Token-Level AFL triggered a similar number of basic blocks as Fuzzilli did, but less than CodeAlchemist. The average number of basic blocks found in each configuration is shown in Table 6.3 and a graph of block coverage over the three

Figure 6.7: This graph shows the block coverage over time that each of the fuzzers had when running on V8. Token-Level AFL was able to continually find and trigger new blocks throughout the three day experiment.

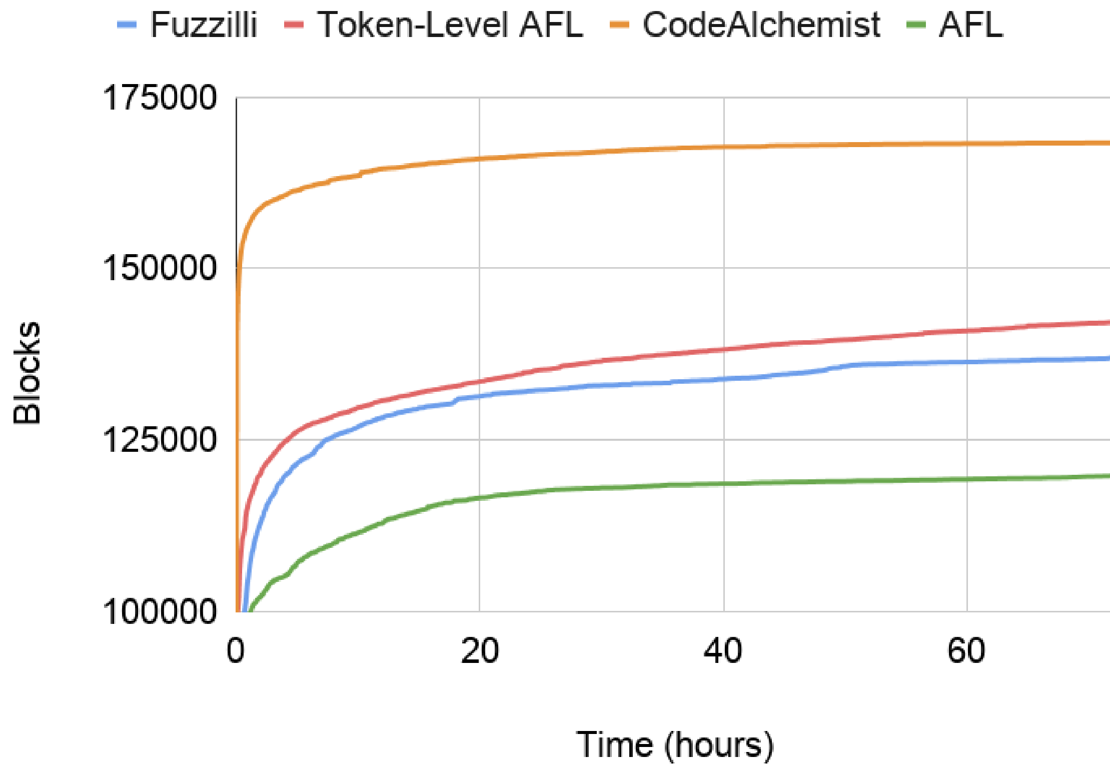


Table 6.3: Average number of basic blocks triggered by each of the tools on each of the target engines. Token-Level AFL performed similarly to Fuzzilli in terms of number of blocks covered. CodeAlchemist, which used many more seeds, had the best block coverage.

	V8	JSC	Spider-Monkey	Chakra-Core
Token-Level AFL	142225	246734	173557	175427
AFL	119847	214790	157881	132134
CodeAlchemist	168340	256665	212495	214401
Fuzzilli	137008	244538	184490	N/A

days of fuzzing is shown in Figure 6.7. When investigating these numbers we discovered that the seeds may play a large role in it. The 32682 seeds given to CodeAlchemist alone triggered about 160000 blocks in V8, whereas the 100 seeds given to Token-Level AFL only triggered about 94000 blocks. Additionally, even with higher coverage, CodeAlchemist triggered much less bugs, showing that code coverage does not yield bugs on its own; assumptions must be violated as well. However, these numbers do show that Token-Level AFL was able to find many blocks that were not triggered by the initial seeds and show that it is competitive in terms of block coverage.

It is worth mentioning that the lack of bugs found by other tools does not necessarily indicate a lack of performance. Instead, it is quite likely that because these fuzzers are open-source they are currently being run and bugs which they can find are reported frequently. These results do show that Token-Level AFL is finding bugs that these tools are not able to find as easily.

Answer for RQ2: Token-Level AFL is able to find bugs which other state-of-the-art fuzzers are unable to. Furthermore, in our tests, it found more bugs in the major JavaScript engines than any of the other fuzzers which we compared it to.

Table 6.4: This table shows the bugs which Token-Level AFL found over a 60 day period of running it against the latest JavaScript engines. Some of these bugs resulted in memory corruption which could lead to exploitation and remote code execution. In the "Status" column we note if we have confirmed that the bug still exists in the most up-to-date code, reported it, or if it was fixed internally. We are currently in the process of responsibly disclosing all confirmed bugs to the respective software vendors.

Id Number	JS Engine	Description	Status	Bug ID
1	V8	Memory corruption while parsing due to parser error.	Reported/Fixed	CR 1015567
2	V8	Debug Check due to incorrect parsing of arrow functions.	Reported/Fixed	V8 9758
3	V8	Null dereference	Fixed Internally	
4	V8	Debug Check in regular expression runtime	Reported/Fixed	CR 1018592
5	V8	Out of bounds indexing in an array due to incorrect parsing	Reported/Fixed	CR 1021457
6	V8	Parser debug check due to incorrectly allocated variable	Fixed Internally	
7	V8	Debug Check in garbage collection	Reported	CR 1044261
8	V8	Debug check when converting integer to index	Fixed Internally	
9	V8	Triggers unreachable code due to frozen elements	Reported/Fixed	CR 1045572
10	V8	Unexpected error handler triggered in JIT	Fixed Internally	
11	V8	Check failed due to incorrect object size	Reported/Fixed	CR 1076106
12	V8	JIT bug leading to memory corruption	Reported	CVE-2020-6468
13	V8	Triggers unreachable code due to frozen elements	Reported	V8 10484
14	V8	Jit bug in bytecode analysis	Fixed internally	
15	V8	Parser error leading to debug check	Confirmed in latest	
16	V8	JIT bug related to a syntax error	Confirmed in latest	
17	JSC	JIT bug resulting in an unexpected switch case	Confirmed in latest	
18	JSC	JIT bug in FTL resulting in an unexpected null pointer	Confirmed in latest	
19	JSC	JIT bug in DFG failing a validation check	Confirmed in latest	
20	JSC	JIT bug in FTL to DFG Lowering	Confirmed in latest	
21	SpiderMonkey	length related assertion	Confirmed in latest	
22	SpiderMonkey	Parser assertion	Confirmed in latest	
23	SpiderMonkey	Parser bug due to unexpected expression	Confirmed in latest	
24	ChakraCore	Type mismatch in parsing	Confirmed in latest	
25	ChakraCore	crash in the JIT	Confirmed in latest	
26	ChakraCore	Array length changed unexpectedly	Confirmed in latest	
27	ChakraCore	Out of Bounds in Array	Confirmed in latest	

6.4.4 Real World Bugs

We have shown that Token-Level AFL is effective for finding bugs in JavaScript engines which other fuzzers are unable to find. These bugs were in the JavaScript engines that were available as of October 1, 2019. Now we want to further explore Token-Level AFL's ability to find real world bugs in the latest engines when run over a long period of time. To do this, we run our fuzzer for 60 days on the latest JavaScript engines. The fuzzer was restarted periodically and JavaScript engines were updated throughout this experiment.

Table 6.4 shows a summary of all the bugs which Token-Level AFL found across JavaScript engines. The table shows which engine each bug was found in and a description of the bug. The status column shows if the bug has been reported by us and fixed. ”Confirmed” indicates that we have confirmed the bug but we have not had time to triage and report it yet. ”Fixed internally” means that they identified and fixed the bug without our report; sometimes these were short-lived bugs.

In total, we found 27 bugs across the major JavaScript engines, demonstrating that Token-Level AFL is capable of finding real world bugs. Note that we found bugs in many areas of the JavaScript engines. These bugs were in various components: from the parser, to regular expressions, to the JIT compilation. We believe this shows not only that Token-Level AFL is capable of finding unknown bugs in JavaScript engines, but also that it is widely applicable and can find bugs in many parts.

Also these bugs include some which are serious, and could lead to remote code execution. We were able to write an RCE exploit for Chrome using bugs which we found with this tool. Furthermore, we have been awarded over ten thousand dollars in bounties, showing the impact of our research.

```
class var6 extends Object {
  constructor ( a,b,c) {
    super (1.1 ) 1 ;
  }
};

new var6();
```

Listing 2: A bug found by Token-Level AFL in V8. This bug was due to a parser error which incorrectly calculated the index into an array. This bug could lead to exploitable memory corruption.

Answer for RQ3: Token-Level AFL is able to find real-world bugs in all of the major JavaScriot engines. This shows this tool has high impact and can be used for finding previously unknown bugs as well as for catching bugs as they are introduced.

6.4.5 Case Study

In this section we investigate some of the bugs to determine if Token-Level AFL is finding bugs that involve invalid syntax, which strict grammar based tools may be unable to find.

In Listing 2, we show (a minimized) example of the a simple bug which Token-Level AFL found. This was a bug in V8 which would lead to memory corruption and possibly could be exploited. This one did require a syntax error to trigger. What happened was new parser code was added that allowed certain incorrect syntax such as the one that is shown. Our tool was able to find this syntax, partially because of its evolutionary behavior. The bug was fixed due to our report and a bounty was awarded.

Another bug we found is shown in Listing 3, which triggered a Debug Check in V8's garbage collector. The code is a minimized version of the real testcase found, after removing redundant statements. This bug is more complex than the previous example, and requires many valid JavaScript statements. We attribute the ability of Token-Level AFL to produce complex valid testcases like these to its coverage guided capabilities. The coverage guided nature will

```
function f () {
  var14=[1,2,3,4,5,6,7,8];
  var15=var14;
  var14.length = 0x100 ;
  var14.__defineGetter__(/./, function(){
    var14.unshift ( 0x20 ) ;
    var14.shift();
    var var3=new Uint32Array(var14);
    Object.entries(var14).toString();
  } ) ;
  print(Object.entries(var14).toString());
}
f();
```

Listing 3: This minimized test case triggers a debug check found in V8. This bug is caused by repeated shifting and unshifting of an array which can trigger a debug check in the garbage collector.

tend to discard testcases which do not hit new functionality, allowing it to explore deep code paths.

Bugs found by our technique included both examples where incorrect syntax or semantics is used to trigger a bug and examples where no such error exists in the test case. Many examples of both are shown in Table 6.4. Also many of the bugs which we found were in the parser, as opposed to the other tools we tested which tends to miss those bugs. Our results show that Token-Level AFL is applicable to finding bugs both in the parser and elsewhere in the JavaScript interpreter.

Answer for RQ4: Bugs found by Token-Level AFL include examples where both entirely valid syntax is used and examples where invalid syntax is needed.

6.5 Discussion

Token-Level Fuzzing is a promising new technique which enables deep fuzzing of JavaScript interpreters, without some of the limitations that come with grammar-based fuzzers. By per-

forming coverage-guided mutations on tokens, rather than individual bytes, it can easily mutate the highly-structured inputs involved in the language. Additionally, because Token-Level Fuzzing is able to find bugs with unusual constructs (syntax and semantics), we believe it will complement the current grammar-based approaches nicely for JavaScript fuzzing. In this section, we will discuss the generalizability of our technique, as well as directions for future work.

6.5.1 Generalizability

Although we implemented and tested Token-Level AFL only on JavaScript engines, the technique is likely applicable to other ”language-based” programs, such as compilers, interpreters and parsers. It would need a different pre-processor, specific to the target, that can separate the text into tokens and identify variables. Similarly, a new decoder would need to be written for that target to transform the encoded input back into the original language. These are not technical challenges, and we believe this technique should be effective on other token-based programs, especially given the results it has shown on JavaScript engines. Furthermore, this is likely easier than adapting a grammar-based fuzzer to a new target.

6.5.2 Seed Selection

Token-Level AFL relies heavily on the input seeds, not just in picking valid tokens, but also the seeds are where the fuzzer begins mutating. It's intuitive that selection can matter greatly. If a seed is close to triggering a bug then the number of mutations needed may be small. In fact, we noticed substantial similarities between some of the bugs that we found and our input testcases which we provided. Additionally, having seeds that trigger a wide variety of functionality helps the fuzzer to explore the various areas of code in the interpreter.

One result shown in Section 6.4 is that Token-Level AFL's block coverage could likely be

improved by having better seeds and giving more seeds to the fuzzer. For our experiments, we used a very ad hoc approach, and our seeds were very incomplete. Applying a better method of seed collection could have a big impact for improving results. For example, Skyfire [105] could be used to generate promising JavaScript seeds. There are also various papers suggesting better seed selection strategies which we could employ to improve our results [106, 107].

6.5.3 Future Work

Because our technique transforms the JavaScript tokens into the familiar binary-based format, we could easily begin applying the various advancements that have been made in the field. For example, because there are so many edges in the JavaScript interpreters, we find that there are many collisions in the edge tracking of AFL. We could use the path sensitivity of CollAFL [108] to help remedy this. Applying ensemble based fuzzing [80], by using Token-Level AFL alongside of a grammar based approach could allow both techniques to build on top of their results. Another direction would be to try to use better prioritization on the inputs as suggested by [109], especially since we typically have tens of thousands of inputs in the fuzzer queue after a few days of fuzzing.

6.6 Conclusion

In this paper, we have presented Token-Level Fuzzing, a new technique for fuzzing language-based programs, such as interpreters. Token-Level Fuzzing allows fuzzing these complex formats without the need of a grammar, allowing it to exercise both the parsing layers as well as the actual interpretation. This relatively simple idea, that we can fuzz at an intermediate level between grammar-based and byte-based fuzzers, provides security researchers a powerful new technique which can be built upon for further research.

In our evaluation Token-Level AFL found 27 new bugs, among which were multiple high-severity issues, across the most up-to-date JavaScript engines. Given the difficulty of fuzzing such programs, we believe this showcases the potential of our technique. Token-Level Fuzzing could show itself to be a powerful new paradigm of fuzzing as security researchers apply it to fuzzing other security critical technologies, and build upon it in future work.

Chapter 7

Related Work

7.1 Driller

7.1.1 Guided Fuzzing

Fuzzing was originally introduced as one of several tools to test UNIX utilities [110]. Since then, it has been extensively used for the black-box security testing of applications. However, fuzzing suffers from a lack of guidance – new inputs are generated based on random mutations of prior inputs, with no control over which paths in the application should be targeted.

The concept of guided fuzzing arose to better direct fuzzers toward specific classes of vulnerabilities. For example, many studies have attempted to improve fuzzing by selectively choosing optimal test cases, honing in on interesting regions of code contained in the target binary [41,42]. Specifically, Dowser [41] uses static analysis to first identify regions of code that are likely to lead to a vulnerability involving a buffer overflow. To analyze this code, Dowser applies taint-tracking to available test cases to determine which input bytes are processed by these code regions and symbolically explores the region of code with only these bytes being symbolic. Unfortunately, Dowser has two drawbacks: it requires test cases to reach the region

of code containing the memory corruption vulnerability, and it only supports buffer overflow vulnerabilities. Unlike Dowser, Driller supports arbitrary vulnerability specifications (though the current implementation focuses on vulnerabilities that lead to a crash) and does not require input test cases. Additionally, Dowser still suffers from the path explosion problem of symbolic execution, while Driller mitigates this problem through its use of fuzzing.

Similar to Dowser, BuzzFuzz [111] applies taint-tracking to sample input test cases to discover which input bytes are processed by 'attack-points' defined by the auditor, most often system call arguments and library code. Unlike BuzzFuzz, Driller does not rely on input test cases that reach vulnerable code, nor does it rely on auditor defined 'attack-points'.

In another attempt to improve the state of fuzzing, Flayer [112] allows an auditor to skip complex checks in the target application at-will. This allows the auditor to fuzz logic deeper within the application without crafting inputs which conform to the format required by the target, at the cost of time spent investigating the validity of crashing inputs found. Similarly, Taintscope uses a checksum detection algorithm to remove checksum code from applications, effectively "patching out" branch predicates which are difficult to satisfy with a mutational approach [113]. This enables the fuzzer to handle specific classes of difficult constraints. Both these approaches, however, either require a substantial amount of human guidance in Flayer's case, or manual effort to determine false positives during crash triaging. Driller does not modify any code of the target application, meaning crashes discovered do not require an in-depth investigation, additionally Driller does not require human intervention, as it attempts to discover well-formed inputs using its concolic execution backend.

Another approach is Hybrid Fuzz Testing, in which limited symbolic exploration is utilized to find "frontier nodes" [114]. Fuzzing is then employed to execute the program with random inputs, which are preconstrained to follow the paths leading to a frontier node. This method is useful for ensuring that the fuzzed inputs take different paths early in the execution of the binary, but it does not handle complex checks, deeper in the program, which separate compart-

ments. Additionally, the path explosion problem effectively prevents the symbolic exploration from solving more than just the shallow checks in the binary.

7.1.2 Whitebox Fuzzing

Other systems attempt to blend fuzzing with symbolic execution to gain maximal code coverage [43, 44, 115, 116]. These approaches tend to augment fuzzing by symbolically executing input produced by a fuzzing engine, collecting symbolic constraints placed on that input, and negating these constraints to generate inputs that will take other paths. However, these tools lack Driller's key insight, that symbolic execution is best used to recover input for driving code execution between application compartments. Without this insight, the unique capabilities of symbolic execution are wasted on creating divergent paths *within* compartments. These tools are, in essence, symbolic execution engines acting in a *serialized* manner, one path at a time, and as such, they are deeply affected by the path explosion problem.

While Driller is similar in a number of implementation details, we propose that we can offload the majority of unique path discovery to an instrumented fuzzing engine. We limit our costly symbolic execution invocations to satisfy conditions that will allow us to enter additional compartments for fuzzing. Since we only use symbolic execution for generating the basic block transitions that the fuzzer has not been able to generate itself, the symbolic execution engine handles a manageable number of inputs. Conversely, the aforementioned tools repetitively negate constraints using concolic execution, slowly analyzing an exponentially increasing number of transitions, most of which can be analyzed more efficiently by a fuzzer.

7.1.3 Concolic Execution

With the continuing increase of computing power in recent years, *concolic execution* (also known as *dynamic symbolic execution*) has risen in popularity. Introduced with EXE [117],

refined with KLEE [45], and applied to binary code with Mayhem [47] and S2E [46], concolic execution engines interpret an application, model user input using symbolic variables, track constraints introduced by conditional jumps, and use constraint solvers to create inputs to drive applications down specific paths. While these systems are powerful, they suffer from a fundamental problem: if a conditional branch depends on symbolic values, it is often possible to satisfy *both* the taken and non-taken condition. Thus, the state has to *fork* and both paths must be explored. This quickly leads to the well-known path explosion problem, which is the primary inhibitor of concolic execution techniques.

Various approaches have been attempted to mitigate the path explosion problem. Veritesting [50] proposed an advanced path merging technique to reduce the number of paths being executed, Firmalice [37] performs extensive static analysis and limits symbolic execution to small slices of code, and under-constrained symbolic execution exchanges precision for scalability [69, 118]. However, these techniques either fail to mitigate the path explosion problem (Veritesting delays the explosion, but such explosion still eventually occurs) or produce inputs that are not directly actionable (for example, the slicing done by Firmalice produces inputs that satisfy the constraints of a particular slice, but no input is provided to *reach* the code in the first place).

Driller attempts to mitigate this by offloading most of the path exploration task to its fuzzing engine, using concolic execution only to satisfy complex checks in the application that guard the transition between compartments.

7.2 Piston

Piston leverages many binary analysis techniques to analyze an executable, determine how to remotely apply the patch, exploit, and repair the remote process. In this section, we will detail work that proposed the program analysis techniques that we use in our system, and

frame Piston in relation to other hot-patching techniques.

7.2.1 Hot-patching

Piston's core contribution is in extending the concept of hot patching to *remote* systems. This can include, like in our evaluation, remote user-space processes but, additionally, could include internet-connected embedded devices that may otherwise not have an update functionality.

Before Piston, hot patching techniques, or *dynamic software updating* approaches have been constrained to patching local processes, often with explicit support from the host system. Originally designed to patch small C programs, they have scaled up to the ability to patch the Linux kernel [119–123]. However, aside from being reliant on source code, these approaches require administrative access to the host machine, which is often unavailable.

To reduce the difficulty of and level of access required by hot-patching systems, techniques have been developed to include hot-patching support in the applications themselves. These systems, which are available for both user-space software [124, 125] and embedded device firmware [58, 126, 127], ease the administrative requirement, but still require pre-planning to include this functionality.

One hot-patching system, ClearView [60], is worth mentioning as it works by monitoring binary code, detecting when it is being exploited, and automatically generating and applying defensive patches. In the latter step, ClearView attempts to *repair* the state of the exploited process state by enforcing invariants. The concept of repairing the process state after exploitation is similar between ClearView and Piston. However, unlike ClearView, Piston does not require administrative access or, in fact, any presence on the device on which the process that needs patching is running. Piston patches, repairs, and resumes remotely, leveraging an exploit to achieve access.

Like most hot-patching systems, Piston relies on the analyst to provide a state transition routine when a patch that it is applying would modify data structures in the program. Recently, some work has been done in automatically recovering such a state transition routine [128, 129]. Though current work requires access to source code (which Piston does not have), a future extension of these techniques to binary code would increase the range of patches that Piston can automatically apply.

Exploit writers targeting operating system kernels have also found themselves repairing state of various parts of memory to allow the kernel to continue running after their exploit payload has been run. This is similar to the recovery and rollback routines used by Piston, but kernel exploiters have done this in a manual, ad hoc manner [130].

7.2.2 Code Injection

Piston patches binaries by injecting new code into the running process. The concept of injecting code at runtime with an exploit has been explored before, albeit not for patching purposes.

Windows malware often achieves code injection by inserting a DLL into the memory of the victim process [131]. This is done to add malicious functionality to a local process. However, this is done *locally*, as opposed to Piston's *remote* code injection, and cannot be done through an exploit. To our knowledge, Piston is the first approach that can inject its code *remotely*, via an exploit, and repair the damage caused by that exploit so that the application can continue.

7.2.3 Analysis Techniques

We utilize many existing binary analysis techniques to build Piston. However, we claim no advancement in the base of binary analysis: Piston's contribution is in the application of binary analyses to remote hot-patching, in composing known analysis techniques in a novel way.

First, we use binary diffing techniques to identify what needs to be updated between the original and the replacement binary. This field has been extensively researched, and many approaches exist for identifying differences in executables, both statically [64, 132–134] and dynamically [135]. While we leverage diffing to determine what patches to apply to the remote process, diffing has also been used for everything from bug searching [136] to automatic exploit generation [137].

Once it determines the patches to apply, Piston uses program analysis techniques to create its repair plan. This includes a type of symbolic execution called under-constrained symbolic execution [69], which extends classical dynamic symbolic execution techniques [45–47] to work on isolated functions in a program. Additionally, we use static analysis techniques to recover the control flow graph of individual functions and to reason about data dependencies between stack variables. We leverage an open-source binary analysis framework, angr[1] [37] for this, which, in turn, uses several static analyses to recover control flow [138–142], identify variables [66], and determine data dependencies [143–145].

7.3 Abstraction Functions in Fuzzing

Fuzzing is a well-known technique for program testing by generating random data as *input* to programs under test, and has drawn much research attention over a wide span of time. The main goal of fuzzing techniques is to violate implicit expectations made by the developer on the input and expose resulting security flaws or bugs.

Input generation: This defines how the inputs are generated by the fuzzing technique. Most of the research in fuzzing occurs in this aspect. There are many well-known input generation techniques:

[1] Available at `https://github.com/angr/angr`

Mutation-based: Here, the fuzzer starts with some seed inputs and new inputs are generated by mutating certain regions of existing inputs [29].

Evolution-based: In this case, evolutionary techniques are used to combine interesting inputs to generate new inputs [19]. We formally define the definition of *interesting* in Section 5.2.

Grammar-based: The fuzzer generates input that satisfies a specified grammar. Program that expect input to conform to a grammar, such as interpreters, and file editors, are generally fuzzed with this input generation technique. If the input structure can be specified as a grammar, then grammar-based techniques can be effective in triggering complex behavior in the target program.

Note that these techniques are not exclusive—tools, such as Dowser [76], combine taint tracking and symbolic execution to generate interesting inputs to trigger buffer overflows. Similarly, Driller [33] combines three strategies: mutation, evolution, and symbolic execution.

Input selection or evaluation: A fuzzing technique must know the effectiveness of its generated inputs, so that it can determine if a new input or mutation strategy was useful. Measuring the effectiveness needs visibility into the program under test and, as such, black-box techniques can only have a limited input evaluation by examining the output of the program. In Section 5.2 we formally redefine input selection and input evaluation as abstraction functions, however for now we will use the informal terms. There are two well-known input evaluation techniques:

Goal-based or directed: Here the input is evaluated on the likelihood of achieving a goal or causing program to reach a certain state. Dowser [76] and BuzzFuzz [146] generate inputs that are likely to cause buffer overflows. libFuzzer-gv [89] explored guiding the fuzzer based on the stack depth and on the number of memory allocations made. How-

ever, these input selection strategies are specialized for each goal and cannot be generalized.

Coverage-based: Here the input is evaluated based on what code is triggered by it on the test program. The intuition is simple: *dynamic techniques cannot find a bug if they do not execute the code containing the bug*, and thus, a higher code coverage implies a higher chance of bugs. Most general fuzzing techniques, such as AFL [29], VUzzer [19], and syzkaller [88] are coverage based.

Most existing research in fuzzing attempts to find new input generation techniques to efficiently generate effective inputs. However, the importance of input selection and evaluation in fuzzing remains an under-explored area and a promising research direction. Wang et al [81] take a step down that road, exploring the differences in some selection strategies. Our work (done independently and concurrently) further explores the effectiveness of various abstraction functions, evaluating a different set of fuzzing strategies, in a larger experiment, with a thorough investigation into the effectiveness of the various strategies.

7.4 Token-Level Fuzzing

Fuzzing is one of the most effective and scalable vulnerability discovery solutions. Fuzzers generate a vast number of test cases to test target applications and monitor their runtime execution to discover unintended security bugs. Most fuzzing research can be divided into three main categories: Input Generation. Program Access. Coverage goals.

Input Generation. There are two main classes of how a fuzzer produces inputs: mutational fuzzing, and generational fuzzing. Mutational fuzzing [19,99,147] modifies one or many seeds of typically well-formed inputs to generate new inputs. Whereas, generational fuzzing tends to be more structure-aware and needs to understand the input format, then it proceeds to generate

inputs following that structure [95, 148, 149].

Program Access. Whitebox fuzzing attempts to do advanced program analysis and collect constraints from conditional branches while execution. Solutions obtained from solving these constraints using constraint solver are then mapped to new inputs [33, 44]. Whereas blackbox fuzzing does not have any access to the internals of the application being tested [150, 151]. Then in the middle is greybox fuzzing, which uses lightweight techniques to gather information such as branch coverage [152].

Coverage Goals. Directed fuzzing has the objective of targeting a set of deep paths for optimization [153]. Coverage-based fuzzing uses different types of tracking such as block coverage, edge coverage, etc. These are used to track the interesting inputs and keep them around for further mutations [21, 29, 88].

7.4.1 Evolutionary Fuzzing

American Fuzzy Lop (AFL) is a security-oriented grey-box fuzzer that does compile-time instrumentation [29]. It has been significantly used to find vulnerabilities and other interesting bugs in many applications [154]. Whenever AFL comes across an input that discovers a new path, it keeps that input around and mutates it further to see if it can hit a new basic block. One of the most promising features of AFL is evolutionary fuzzing which uses coverage based fuzzing to generate new inputs by evolving the promising current inputs. Whenever an input triggers a new edge in the program that input would be added to the queue to fuzz further. An efficient evolutionary fuzzer should be able to reach high coverage by using the feedback from each test case to evolve better test cases that cover the majority of the program code.

There has been much work on improving evolutionary fuzzing. For example, Vuzzer [19] focuses on extracting two main features namely data-flow features (using taint analysis) and control flow features to make a smart feedback loop. These features help infer important prop-

erties of input and prioritize/ deprioritize certain paths. Vuzzer uses static analysis to help with the process of feature extraction [19]. AFLFAST, on the other hand, uses a Markov Chain search strategy to choose low-frequency paths enabling it to explore more paths in the same fuzzing time [155]. Another approach is Angora [21]. Angora uses byte-level taint tracking and gradient-based search algorithm as well as input length exploration and context-sensitive branch count.

7.4.2 JavaScript Fuzzing

Javascript engines are one of the most complicated components of modern-day browsers making it a very popular target for attackers. There have been continuous efforts towards improving the fuzzers to find JavaScript engine vulnerabilities. Some of the most popular JavaScript fuzzers have been centered around generating syntactically correct test cases based on either a predefined context-free grammar or a trained probabilistic language model. JS-FunFuzz is one such JS grammar-based fuzzer that relies purely on the generative approach to create new test-cases [156]. It has been widely used and covers a wide range of JavaScript language features. Another example of a generative approach is Domato [157], which uses HTML, CSS, and JavaScript grammar to generate samples that target DOM specific logic.

Coverage-guided fuzzing has also been successful in finding JavaScript engine vulnerabilities. One of the most common targets for mutation is JS Abstract Syntax Tree [92, 94]. Fuzzilli [99] developed an intermediate language called FuzzIL and the mutations are defined on it for better control and data flows while fuzzing. Fuzzili pivoted on the idea of generating semantically valid programs in high numbers to avoid the need for try-catch constructs.

Interestingly, CodeAlchemist [95] presents an alternative idea to use semantics-aware assembly to produce JavaScript code snippets. They break JS seeds into code fragments and each fragment is tagged with constraints and analyzed for used variables. The code fragments

are then combined to produce syntactically and semantically correct test-cases. LangFuzz [96] also employed the concept of code fragments combined with generative and mutation-based fuzzing to maintain the syntax and semantics of code samples. One key feature of LanFuzz is that its language-independent, which means that it maintains its testing strategy solely on grammar and existing programs and not language-specific information.

Another state-of-the-art work on JS fuzzing uses the concept of aspect preserving mutation [158]. The goal of this technique is to keep beneficial properties from the original seed and retain them across mutations. For example, control-flow structures like loops can trigger JIT compilation and could find a buggy optimization logic. By preserving such aspects, the properties to trigger vulnerabilities can remain, allowing the fuzzer to find more vulnerabilities.

Chapter 8

Looking Forward

Throughout my graduate studies, I have tried to examine problems in unique ways. In my work on exploring abstraction functions, I identified a major component of fuzzing that had been largely ignored in research, and I showed how by using different strategies we can find more bugs. In Token-Level fuzzing, I have introduced a new paradigm of fuzzing for interpreters, which differs from all the previous approaches, yet shows the ability to do as well as state of the art grammar-based fuzzers.

One major area of work which I would like to explore more in my future research is how to include feedback from humans into automated bug hunting techniques. In my work so far, I have focused mainly on fully automated techniques. Of course, there are many situations where humans could be in the loop, both assisted by and assisting the tools that they use. In industry, humans are monitoring and working with bug-hunting techniques all the time in order to eliminate bugs in security-critical software. As such, I think that we can use human effort to guide analyses, as well as making analyses that will guide humans.

Another future research direction is to explore how we can guide fuzzers towards finding bugs, rather than finding more code coverage. My work in Chapter 5, was a first stab at discovering different metrics for fuzzer exploration. However, that research only touches the surface

of an area which might hold great potential for improving bug-finding capabilities.

There are many unsolved issues in vulnerability discovery, and many advancements will be needed before we can reach an end goal of having no exploitable vulnerabilities remaining. I hope this thesis can be a catalyst for future researchers and that they will approach problems in a unique way when building upon the work done here.

Bibliography

[1] Google, "Chromium github." `https://github.com/chromium/chromium`.

[2] Apple, "Webkit github." `https://github.com/WebKit/webkit`.

[3] "Firefox github, author=Mozilla, note=`https://github.com/mozilla/gecko-dev`."

[4] Microsoft, "Edge." `https://github.com/MicrosoftEdge`.

[5] Linux, "Linux github." `https://github.com/torvalds/linux`.

[6] Microsoft, "Windows technet." `https://docs.microsoft.com/en-us/previous-versions//cc767881(v=technet.10)`.

[7] Apple, "Osx wiki." `https://en.wikipedia.org/wiki/MacOS`.

[8] I. Fratric, "Bypassing mitigations by attacking jit server." `https://raw.githubusercontent.com/google/p0tools/master/JITServer/JIT-Server-whitepaper.pdf`.

[9] B. Sun and C. Xu, "Bypassing memory mitigations using data only." `https://conference.hitb.org/hitbsecconf2017ams/sessions/bypassing-memory-mitigations-using-data-only-exploitation-techniques-part-ii/`.

[10] B. Azad, "Examining pointer authentication on iphone xs." `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`.

[11] B. Kirkpatrick, *Computer security is algorithmically intractable*, .

[12] C. Salls, Y. Shoshitaishvili, N. Stephens, C. Kruegel, and G. Vigna, *Piston: Uncooperative remote runtime patching*, in *Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 141–153, 2017.

[13] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, *Katana: A hot patching framework for elf executables*, in *2010 International Conference on Availability, Reliability and Security*, pp. 507–512, IEEE, 2010.

[14] R. Langner, *Stuxnet: Dissecting a cyberwarfare weapon*, *IEEE Security & Privacy* **9** (2011), no. 3 49–51.

[15] C. honzalez, "Software updates are the new hurdle in iot security." `https://www.machinedesign.com/automation-iiot/article/21836333/software-updates-are-the-new-hurdle-in-iot-security`.

[16] A. Green, "The mirai botnet attack." `https://www.varonis.com/blog/the-mirai-botnet-attack-and-revenge-of-the-internet-of-things/`.

[17] M. Bhme, V. Pham, and A. Roychoudhury, *Coverage-based greybox fuzzing as markov chain*, *IEEE Transactions on Software Engineering* **45** (2019), no. 5 489–506.

[18] W. S. ChangwooMin and T. Kim, *Designing new operating primitives to improve fuzzing performance*, .

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-aware evolutionary fuzzing*, in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[20] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, *Steelix: Program-state based binary fuzzing*, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), pp. 627–637, ACM, 2017.

[21] P. Chen and H. Chen, *Angora: Efficient fuzzing by principled search*, in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 711–725, IEEE, 2018.

[22] M. Bill and J. Scott-Railton, "The million dollar dissident." `https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/`.

[23] J. J. Rooney and L. N. V. Heuvel, *Root cause analysis for beginners*, *Quality progress* **37** (2004), no. 7 45–56.

[24] A. I. Sotirov, *Automatic vulnerability detection using static source code analysis*. PhD thesis, Citeseer, 2005.

[25] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *{DR}.{CHECKER}: A soundy analysis for linux kernel drivers*, in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1007–1024, 2017.

[26] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, *Bap: A binary analysis platform*, in *International Conference on Computer Aided Verification*, pp. 463–469, Springer, 2011.

[27] K. Chen and D. Wagner, *Large-scale analysis of format string vulnerabilities in debian linux*, in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, pp. 75–84, 2007.

[28] G. Balakrishnan and T. Reps, *WYSINWYX: What You See is Not What you Execute*. PhD thesis, University of Wisconsin at Madison, August, 2007.

[29] M. Zalewski., *American fuzzy lop*, 2017. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[30] https://llvm.org/docs/LibFuzzer.html, 2019.

[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et. al.*, *Sok:(state of) the art of war: Offensive techniques in binary analysis*, in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, IEEE, 2016.

[32] P. Godefroid, M. Y. Levin, D. A. Molnar, *et. al.*, *Automated whitebox fuzz testing.*, in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2008.

[33] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*, in *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[34] "Chrome issue 1076708." https://bugs.chromium.org/p/chromium/issues/detail?id=1076708.

[35] Secunia, "Resources vulnerability review 2015." http://secunia.com/resources/vulnerability-review/introduction/.

[36] "Vulnerability distribution of CVE security vulnerabilities by type." http://www.cvedetails.com/vulnerabilities-by-types.php.

[37] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, *Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware*, .

[38] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, *Taming compiler fuzzers*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 48, ACM, 2013.

[39] J. DeMott, "Understanding how fuzzing relates to a vulnerability like Heartbleed." http://labs.bromium.com/2014/05/14/understanding-how-fuzzing-relates-to-a-vulnerability-like-heartbleed/.

[40] S. Bucur, *Improving Scalability of Symbolic Execution for Software with Complex Environment Interfaces*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2015.

[41] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, *Dowsing for overflows: A guided fuzzer to find buffer boundary violations.*, in *Proceedings of the USENIX Security Symposium*, 2013.

[42] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, *The BORG: Nanoprobing binaries for buffer overreads*, in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, ACM, 2015.

[43] P. Godefroid, N. Klarlund, and K. Sen, *DART: Directed automated random testing*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 40, pp. 213–223, ACM, 2005.

[44] P. Godefroid, M. Y. Levin, and D. Molnar, *SAGE: Whitebox fuzzing for security testing*, *Communications of the ACM* **55** (2012), no. 3 40–44.

[45] C. Cadar, D. Dunbar, D. R. Engler, *et. al.*, *Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.*, in *OSDI*, vol. 8, pp. 209–224, 2008.

[46] V. Chipounov, V. Kuznetsov, and G. Candea, *S2E: A platform for in-vivo multi-path analysis of software systems*, vol. 47. ACM, 2012.

[47] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, *Unleashing mayhem on binary code*, in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 380–394, IEEE, 2012.

[48] F. Bellard, *QEMU, a fast and portable dynamic translator*, in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[49] N. Nethercote and J. Seward, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 42, pp. 89–100, ACM, 2007.

[50] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, *Enhancing symbolic execution with veritesting*, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 1083–1094, ACM, 2014.

[51] DARPA, "Cyber Grand Challenge." `http://cybergrandchallenge.com`.

[52] LegitBS, "DEFCON Capture the Flag." `https://legitbs.net/`.

[53] DARPA, "Cyber Grand Challenge Challenge Repository." `https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges`.

[54] P. Garg, “Fuzzing - mutation vs. generation.”
`http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/`.

[55] R. van der Meulen, “Gartner says 6.4 billion connected ”things” will be in use in 2016, up 30 percent from 2015.”
`http://www.gartner.com/newsroom/id/3165317`.

[56] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, *The attack of the clones: a study of the impact of shared code on vulnerability patching*, in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 692–708, IEEE, 2015.

[57] M. A. McQueen, T. A. McQueen, W. F. Boyer, and M. R. Chaffin, *Empirical estimates and observations of 0day vulnerabilities*, in *System Sciences, 2009. HICSS’09. 42nd Hawaii International Conference on*, pp. 1–12, IEEE, 2009.

[58] H. Martorell, J.-C. Fabre, M. Roy, and R. Valentin, *Towards dynamic updates in autosar*, in *SAFECOMP 2013-Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*, p. NA, 2013.

[59] Oracle, “Ksplice.” `http://www.ksplice.com/`.

[60] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, *et. al.*, *Automatically patching errors in deployed software*, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 87–102, ACM, 2009.

[61] D. Goodin, *Windows 7, not xp, was the reason last weeks wcry worm spread so widely*, 2017.
`https://arstechnica.com/security/2017/05/windows-7-not-xp-was-the-reason-last-weeks-wcry-worm-spread-so-widely/`.

[62] F. Castaneda, E. C. Sezer, and J. Xu, *Worm vs. worm: preliminary study of an active counter-attack mechanism*, in *Proceedings of the 2004 ACM workshop on Rapid malcode*, pp. 83–93, ACM, 2004.

[63] “Cve-2013-2028 advisory.” `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2028`.

[64] T. Dullien and R. Rolles, *Graph-based comparison of executable objects (english version)*, *SSTIC* **5** (2005) 1–3.

[65] Y.-Y. Chang, P. Zavarsky, R. Ruhl, and D. Lindskog, *Trend analysis of the cve for software vulnerability management*, in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, pp. 1290–1293, IEEE, 2011.

[66] J. Lee, T. Avgerinos, and D. Brumley, *TIE: principled reverse engineering of types in binary programs*, in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[67] M. Noonan, A. Loginov, and D. Cok, *Polymorphic type inference for machine code*, in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 27–41, ACM, 2016.

[68] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Steal this movie: Automatically bypassing drm protection in streaming media services.*, in *USENIX Security*, pp. 687–702, 2013.

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

[70] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, *Patchdroid: Scalable third-party security patches for android devices*, in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 259–268, ACM, 2013.

[71] "Nginx cve 2013-2028 patch." `http://nginx.org/download/patch.2013.chunked.txt`.

[72] G. McManus, hal, and saelo, "Nginx cve 2013-2028 metasploit exploit." `https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/linux/http/nginx_chunked_size.rb`.

[73] "Nginx cve 2013-2028 kingcope exploit." `https://www.exploit-db.com/exploits/26737/`.

[74] J. Drake, *Stagefright: Scary code in the heart of android*, *BlackHat USA* (2015).

[75] P. Cousot and R. 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*, pp. 238–252, ACM, 1977.

[76] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, *Dowser: a guided fuzzer to find buffer overflow vulnerabilities*, in *Proceedings of the 22nd USENIX Security Symposium*, pp. 49–64, 2013.

[77] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, *Difuze: Interface aware fuzzing for kernel drivers*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, ACM, 2017.

[78] D. Vyukov, *Kcov: Kernel coverage*, 2017.
https://lwn.net/Articles/671640/.

[79] CVE-2013-0997. https://packetstormsecurity.com/files/123229/Apple-Security-Advisory-2013-09-12-2.html, 2013.

[80] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, *Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers*, in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1967–1983, 2019.

[81] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, *Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing*, in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pp. 1–15, 2019.

[82] DARPA, *Darpa cyber grand challenge*, 2016.
http://archive.darpa.mil/cybergrandchallenge/.

[83] LungeTech, *Cgc data archive for qualifiers*, 2017.
http://www.lungetech.com/cgc-corpus/cwe/cqe/.

[84] LungeTech, *Cgc data archive for finals*, 2017.
http://www.lungetech.com/cgc-corpus/cwe/cfe/.

[85] T. of Bits, *Darpa challenge binaries on linux, os x, and windows*, 2017.
https://github.com/trailofbits/cb-multios.

[86] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, *Evaluating fuzz testing*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, ACM, 2018.

[87] G. McGraw, S. Migues, and J. West, *Bsimm8*, 2017.
https://www.bsimm.com/content/dam/bsimm/reports/bsimm8.pdf.

[88] Google, *syzkaller - linux syscall fuzzer*, 2017.
https://github.com/google/syzkaller.

[89] G. Vranken, "libfuzzer-gv: new techniques for dramatically faster fuzzing."
https://guidovranken.wordpress.com/2017/07/08/libfuzzer-gv-new-techniques-for-dramatically-faster-fuzzing/, 2017.

[90] Google. https://google.github.io/oss-fuzz/getting-started/continuous-integration.

[91] P. Godefroid, A. Kiezun, and M. Y. Levin, *Grammar-based whitebox fuzzing*, in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 206–215, 2008.

[92] J. Wang, B. Chen, L. Wei, and Y. Liu, *Superion: Grammar-aware greybox fuzzing*, in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 724–735, IEEE, 2019.

[93] A. LEE, *Fuzzing javasscript engines for fun and pwnage*, 2018.

[94] R. Guo, *Mongodb's javascript fuzzer*, *Queue* **15** (2017), no. 1 38–56.

[95] H. Han, D. Oh, and S. K. Cha, *Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines.*, in *NDSS*, 2019.

[96] C. Holler, K. Herzig, and A. Zeller, *Fuzzing with code fragments.*, in *Proceedings of the USENIX Security Symposium*, pp. 445–458, 2012.

[97] "Peach." `https://www.peach.tech/`.

[98] J. Pereyda, "boofuzz." `https://github.com/jtpereyda/boofuzz`.

[99] S. Groß, *FuzzIL: Coverage Guided Fuzzing for JavaScript Engines*. PhD thesis, TU Braunschweig, 2018.

[100] *Issue 800032: Security: V8: Bugs in genesis::initializeglobal*, 2018. `https://bugs.chromium.org/p/chromium/issues/detail?id=800032`.

[101] Google. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1308`.

[102] Michal Zalewski, *afl-fuzz: making up grammar with a dictionary in hand*, 2015. `https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html`.

[103] Engadget, *Microsoft's chromium edge browser arrives january 15th*, 2019. `https://www.engadget.com/2019-11-04-chromium-edge-browser-release-date.html`.

[104] The Chromium Project. `https://chromium.googlesource.com/chromium/src/+/master/styleguide/c++/c++.md#CHECK_DCHECK_and-NOTREACHED`.

[105] J. Wang, B. Chen, L. Wei, and Y. Liu, *Skyfire: Data-driven seed generation for fuzzing*, 2017.

[106] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, *Optimizing seed selection for fuzzing*, in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 861–875, USENIX Association, 2014.

[107] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, *Optimizing seed inputs in fuzzing with machine learning*, in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 244–245, IEEE, 2019.

[108] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, *Collafl: Path sensitive fuzzing*, in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 679–696, IEEE, 2018.

[109] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, *Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization*, in *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2020.

[110] B. P. Miller, L. Fredriksen, and B. So, *An empirical study of the reliability of unix utilities*, *Communications of the ACM* **33** (1990), no. 12 32–44.

[111] V. Ganesh, T. Leek, and M. Rinard, *Taint-based directed whitebox fuzzing*, in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2009.

[112] W. Drewry and T. Ormandy, *Flayer: Exposing application internals*, in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[113] T. Wang, T. Wei, G. Gu, and W. Zou, *Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection*, in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 497–512, IEEE, 2010.

[114] B. S. Pak, *Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution*, Master's thesis, School of Computer Science, Carnegie Mellon University, May, 2012.

[115] G. Campana, *Fuzzgrind: un outil de fuzzing automatique*, *Actes du* (2009) 213–229.

[116] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song, *Transformation-aware exploit generation using a HI-CFG*, tech. rep., UCB/EECS-2013-85, 2013.

[117] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, *EXE: Automatically generating inputs of death*, *ACM Transactions on Information and System Security (TISSEC)* **12** (2008), no. 2 10.

[118] D. Engler and D. Dunbar, *Under-constrained execution: Making automatic code destruction easy and scalable*, in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2007.

[119] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, *Practical dynamic software updating for C*, vol. 41. ACM, 2006.

[120] A. Sotirov, “Hotpatching and the rise of third-party patches.” BlackHat USA, 2006.

[121] I. Neamtiu and M. Hicks, *Safe and timely updates to multi-threaded programs*, in *ACM Sigplan Notices*, vol. 44, pp. 13–24, ACM, 2009.

[122] M. Siniavine and A. Goel, *Seamless kernel updates*, in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2013.

[123] J. Arnold and M. F. Kaashoek, *Ksplice: Automatic rebootless kernel updates*, in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 187–198, ACM, 2009.

[124] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, *State transfer for clear and efficient runtime updates*, in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pp. 179–184, IEEE, 2011.

[125] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, *Kitsune: Efficient, general-purpose dynamic software updating for c*, in *ACM SIGPLAN Notices*, vol. 47, pp. 249–264, ACM, 2012.

[126] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, *Safe and automatic live update for operating systems*, *ACM SIGPLAN Notices* **48** (2013), no. 4 279–292.

[127] H. Martorell, J.-C. Fabre, M. Roy, and R. Valentin, *Improving adaptiveness of autosar embedded applications*, in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 384–390, ACM, 2014.

[128] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, *Dynamic software updating using a relaxed consistency model*, *Software Engineering, IEEE Transactions on* **37** (2011), no. 5 679–694.

[129] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, *Back to the future: Fault-tolerant live update with time-traveling state transfer.*, in *LISA*, pp. 89–104, 2013.

[130] E. Perla and M. Oldani, *A guide to kernel exploitation: attacking the core*. Elsevier, 2010.

[131] Wikipedia, “Dll injection.” `https://en.wikipedia.org/wiki/DLL_injection`.

[132] M. Bourquin, A. King, and E. Robbins, *Binslayer: accurate comparison of binary executables*, in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, p. 4, ACM, 2013.

[133] H. Flake, *Structural comparison of executable objects*, .

[134] M. Bourquin, A. King, and E. Robbins, *Accurate comparison of binary executables*, .

[135] M. Egele, M. Woo, P. Chapman, and D. Brumley, *Blanket execution: Dynamic similarity testing for program binaries and components*, in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 303–317, 2014.

[136] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, *Cross-architecture bug search in binary executables*, in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 709–724, IEEE, 2015.

[137] D. Brumley, P. Poosankam, D. Song, and J. Zheng, *Automatic patch-based exploit generation is possible: Techniques and implications*, in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 143–157, IEEE, 2008.

[138] L. Xu, F. Sun, and Z. Su, *Constructing precise control flow graphs from binaries*, *University of California, Davis, Tech. Rep* (2009).

[139] C. Cifuentes and M. Van Emmerik, *Recovery of jump table case statements from binary code*, in *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pp. 192–199, IEEE, 1999.

[140] J. Troger and C. Cifuentes, *Analysis of virtual method invocation for binary translation*, in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pp. 65–74, IEEE, 2002.

[141] B. Schwarz, S. Debray, and G. Andrews, *Disassembly of executable code revisited*, in *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pp. 45–54, IEEE, 2002.

[142] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, *Static disassembly of obfuscated binaries*, in *USENIX security Symposium*, vol. 13, pp. 18–18, 2004.

[143] G. Balakrishnan and T. Reps, *WYSINWYX: What you see is not what you execute*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **32** (2010), no. 6 23.

[144] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, *Signedness-agnostic program analysis: Precise integer bounds for low-level code*, in *Programming Languages and Systems*, pp. 115–130. Springer, 2012.

[145] Tok, Teck Bok and Guyer, Samuel Z and Lin, Calvin, *Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers*, in *Compiler Construction*, pp. 17–31, Springer, 2006.

[146] V. Ganesh, T. Leek, and M. Rinard, *Taint-based directed whitebox fuzzing*, in *Proceedings of the 31st International Conference on Software Engineering*, ICSE ’09, (Washington, DC, USA), pp. 474–484, IEEE Computer Society, 2009.

[147] S. K. Cha, M. Woo, and D. Brumley, *Program-adaptive mutational fuzzing*, in *2015 IEEE Symposium on Security and Privacy*, pp. 725–741, IEEE, 2015.

[148] H. Han and S. K. Cha, *Imf: Inferred model-based fuzzer*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2345–2358, 2017.

[149] G. Grieco, M. Ceresa, and P. Buiras, *Quickfuzz: An automatic random fuzzer for common file formats*, *ACM SIGPLAN Notices* **51** (2016), no. 12 13–20.

[150] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, *Enemy of the State: A State-Aware Black-Box Vulnerability Scanner*, in *Proceedings of the USENIX Security Symposium*, Aug., 2012.

[151] D. Wang, X. Zhang, T. Chen, and J. Li, *Discovering vulnerabilities in cots iot devices through blackbox fuzzing web management interface*, *Security and Communication Networks* **2019** (2019).

[152] A. Zeller, R. Gopinath, M. Bhme, G. Fraser, and C. Holler, *Greybox fuzzing*, 2019. `https://www.fuzzingbook.org/html/GreyboxFuzzer.html`.

[153] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, *Directed greybox fuzzing*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017.

[154] M. Zalewski., *American fuzzy lop*, 2017. `http://lcamtuf.coredump.cx/afl/`.

[155] M. Böhme, V.-T. Pham, and A. Roychoudhury, *Coverage-based greybox fuzzing as markov chain*, *IEEE Transactions on Software Engineering* **45** (2017), no. 5 489–506.

[156] J. Ruderman, *Introducing jsfunfuzz*, 2007. `https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`.

[157] I. Fratric, *The great dom fuzz-off of 2017*, 2017.

[158] S. P. W. X. I. Yun and D. J. T. Kim, *Fuzzing javascript engines with aspect-preserving mutation*, in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.