Towards integrating static code analysis and hybrid fuzzing for more efficient bug detection

Nasir Abdi Awed

Thesis submitted for the degree of
Master in Programming and System Architecture:
Software
60 credits

Institute of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2022

Towards integrating static code analysis and hybrid fuzzing for more efficient bug detection

Nasir Abdi Awed

© 2022 Nasir Abdi Awed

Towards integrating static code analysis and hybrid fuzzing for more efficient bug detection

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Software is not perfect; any software system can be and probably is vulnerable. Perfect security is very much improbable, no matter what you do, how many flaws you patch, or the capital you invest. Finding bugs in software is becoming exceedingly tricky; it takes researchers, developers, and bug hunters endless hours of analyzing through many lines of code. Finding a bug early in production is paramount; conversely, the potential financial cost could be significant if the bug was discovered by triggering an error post-production or a bad actor exploits it.

Automated bug discovery techniques exist, and there exists a plethora of research into such techniques. However, the computational cost of using these techniques is still expensive enough not to be standard practice.

In this thesis, we investigate if it's possible to combine static code analysis with a hybrid fuzzer to improve the efficiency of finding software defects. We will also classify software defects, specifically memory defects, based on observable properties. Next, we perform an empirical evaluation on two datasets containing software defects.

After our empirical evaluation, we see that our direction does aid in discovery time and the need for less exploration; however, not when it comes to the number of bugs exposed. We also see a trend in the class of memory bugs usually found by a hybrid fuzzer.

Integrating static code analysis with a hybrid fuzzer can find software defects more efficiently and could be a valuable tactic for targeted software analysis.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Motivation 2

1.3 Problem statement 3

1.4 Research questions 4

1.5 Research objectives and scope 5

1.6 Research Method 6

1.7 Main Contributions 8

1.8 Thesis Outline 9

2 Background **11**

2.1 DARPA's Cyber Grand Challenge 11

2.2 Fuzzing 13

2.3 AFL: American fuzzy loop 15

2.3.1 AFLplusplus 17

2.4 Symbolic execution 18

2.4.1 Concolic testing 20

2.5 QSYM 21

2.5.1 Bottlenecks in concolic execution addressed by QSYM 21

2.5.2 Satisfiability modulo theories 24

2.6 Static code analysis 25

2.7 Facebook's *infer* 26

2.7.1 Separation logic and bi-abduction 28

2.8 Tactics employed in the CGC 30

2.8.1 Bug discovery 30
2.8.2 From Bug To Vulnerability 32
2.9 Software defect characterization 34
2.9.1 Bug type 35
2.9.2 Evaluate static and dynamic analysis techniques with regard to the memory bug model 40

3 Project Implementation and Evaluation 43
3.1 Planning 43
3.2 Ex3 45
3.2.1 Singularity 45
3.3 Datasets 46
3.3.1 CVE dataset 46
3.3.2 LAVA dataset 49
3.4 Experimental design 52

4 Results 55
4.1 LAVA files 55
4.2 CVE files 58
4.2.1 Answering the Research Questions 59
4.2.2 Research question 1 59
4.2.3 Research question 2 60
4.2.4 Research question 3 62
4.3 limitations 64

5 Conclusion 67
5.1 Summary 67
5.2 Suggested Future Work 68

A Analysis tools 71

List of Figures

2.1 AFL edge coverage [74] 15
2.2 AFL workflow exploring a target program 16
2.3 Symbolic execution tree 19
2.4 QSYM-architecture [73] 23
2.5 Overview of SMT solver architecture 24
2.7 Infer call graph [17] 27
2.6 High level view of infer's workflow [17] 27
2.8 Figure describing the cause consequence classification in [10] 36
2.9 Shows the improper state transitions that leads to the software bug CVE-2017-12762 [22] 39
2.10 Connected state transitions [10] 41

3.1 file CVE bug class [11] 46
3.2 yara CVE bug class [11] 47
3.3 Lepton CVE bug class [11] 48
3.4 Gifsicle CVE bug class [11] 48
3.5 Openjpeg CVE bug class [11] 49
3.6 Overall view of how the tools are used together 52

4.1 *jpegS* dynamic analysis results. 56
4.2 *yamlB4* dynamic analysis results. 57
4.3 Graph representing bug discovery over time for the LAVA-dataset. 62
4.4 Graph depicting difference number of test cases generated under fuzzing. 63

List of Tables

3.1 Table with number of files tested, their CVE or how many synthetic bugs within the source 44

4.1 LAVA experimental results 55

4.2 Bug class distribution 61

4.3 LAVA results (note that multiple inputs can crash on the same LAVA bug, creating duplicates) and CVE results 63

Acknowledgements

I like to express my gratitude to my family, friends, and all those in between. The words of wisdom, affirmations, and support have been invaluable to me. Special thanks to my supervisor Leon Moonen for the valuable feedback and guidance you gave me throughout this work. I would also express my gratitude to my co-supervisors Omar Al-Bataineh (Simula), and Gudmund Grov (IFI).

Terminology

Fuzzing: Fuzzing is a software testing technique to validate software by randomizing inputs to a given program [48]. The program's execution is monitored, too see if the randomized input reveals any bugs or software vulnerability. We will describe fuzzing in greater detail in the background chapter of this thesis.

Bug: A bug is a software or hardware defect that causes unintended behavior of a program. A bug might be a vulnerability, or it might not be. *Bug* and *software defect* will be used interchangeably throughout this thesis.

Symbolic execution: Symbolic execution is a technique to analyze a program where the entire execution paths are evaluated symbolically, and each program path can be evaluated[40]. This will also be explored in much greater detail in the backgrounder chapter of this thesis. ***Concolic execution*** is the simplification and more efficient variant of symbolic execution, mixing of **conc**rete and symb**olic** execution of a given software; again, this will be explained in the background chapter.

Hybrid fuzzer: A hybrid fuzzer combines fuzzing and some analysis technique(s), for example symbolic execution, to guide the generation of inputs and make the fuzzing more effective. Again, this will be evaluated in greater detail later in this thesis.

Chapter 1

Introduction

1.1 Context

One might argue that humanity has a vested interest in keeping the digital world equally secure as the physical one. The latter has quite literally thousands of years as a head start but still cannot guarantee perfect security. So it is not surprising that the digital world has a long way to go.

Our financial system, food supply chain, electrical grid, and so much more are dependent on our digital systems' security. Most people will not even think about digital security until something happens, like the Covid-19 pandemic. The Cybersecurity and Infrastructure Security Agency issued an alert early in the pandemic [28], in cooperation with the US department of homeland security and the UK's national cybersecurity center. They argued that the shift from office work to working digitally during the pandemic would escalate the use of vulnerable services. There is also anecdotal evidence that links increased cyber attacks as a consequence of the pandemic [46]. The pandemic exposed our need for reliable systems and showed how fast a world tragedy could be monopolized by malicious actors. Pandemic aside, there are indications that major events do indeed affect cyber-security according to Lallie et al. [43].

Cybersecurity readiness is crucial, but it is not enough to be secure

oneself; we depend on other people's investment in security. The recent SolarWinds hack showed us that [55]. Readiness demands talented people. That demand is currently not satisfied. The world is suffering from a shortfall of professionals to deal with the number of threats we face. 3,790 individuals that work in security responded to the $(ISC)^2$ cybersecurity workforce study [1]. More than half (56 percent) have the opinion that the shortage of talent is risking the safety of their organizations.

1.2 Motivation

The cybersecurity threat is huge, and the cybersecurity industry is still young; we are still learning how to work jointly across countries and businesses. In addition, there are challenges in sharing information; some countries consider IP addresses personal information while others do not. Other reasons to not share information are the perceived reputation loss accompanying the aftermath of a cyber intrusion; an unwillingness to help rival companies keep a competitive edge by increasing their security. Many factors will prevent businesses from sharing information about cyberattacks [66].

These hurdles help malicious actors and prevent professionals from gaining valuable intel. Attackers are writing sophisticated viruses and malware or using zero-day attacks. All of this takes plenty of personnel resources along with an accurate view of the threat landscape to handle, and the industry has neither. An automated cybersecurity system might be our best hope. A system that can deal with code obfuscation, reason about run-time errors, or logical security holes will probably save the companies billions and consumers their privacy.

In 2016, Defense Advanced Research Projects Agency (DARPA) attempted to address the issue mentioned above. They launched the cyber grand challenge (CGC), a competition where automation was applied to software analysis and repair.

The competition also focused on removing the human element as

the principal defense mechanism. Instead, competitors built a cyber reasoning system (CRS) to handle vulnerability discovery and repair of said vulnerability. In other words, this was an attempt to leverage a machine's speed and efficiency while at the same time attempting to implement reasoning abilities.

Automated vulnerability discovery is a central part of a system for bug handling, and enhancing that is this thesis's primary focus and motivation.

1.3 Problem statement

During the competition of the CGC, the teams developed powerful tools to find exploitive bugs in a program. The competition's goal was to automate bug discovery and repair bugs that can be vulnerable. This thesis will focus on discovering bugs in a program, using the techniques employed by the competitors of this competition, and leveraging static code analysis.

Almost all teams used two main approaches, fuzzing and symbolic/concolic execution, combining them into a hybrid fuzzer. Most of the crashing inputs were found by leveraging both methods in tandem. The fuzzer mutates and generates test cases as seeds for the symbolic execution to trace the execution path and overcome conditions that the fuzzer cannot find appropriate inputs for in a reasonable time; in turn, the symbolic execution is limited to the test cases generated by the fuzzer.

Now, they caught some bugs because the program crashed by fuzzing and some bugs caught because the program crashed because of the symbolic execution. Why is that? Is it the way they integrated the tools? or are some bugs inherently easier to catch using one method over the other?

Resources is another issue in automating bug discovery in a program; in theory, if you had infinite computational power, we could leverage a symbolic execution engine to explore all paths in a given program. But, alas, reality forces us to follow the laws of physics and find ways to use what we have as efficiently as possible, and it was the same in the cyber grand challenge(CGC). However, the CGC also benefited from a

competition framework, using resources that are not economically feasible in real-world software. The servers had 1,280 physical cores, 16 TB of memory, and 64 TB of disk space [61]. In addition, 300 kW was used to power the monstrous servers and water-cooled to handle the heat generated. Our focus is on bug discovery; this might reduce resource concerns. However, we will evaluate the fuzzing and symbolic execution techniques with static code analysis to see if we can increase the number of bugs using a highly scalable technique as static code analysis without increasing our resource consumption too much.

We choose to theoretically and empirically evaluate each of the techniques to answer the issues raised in this section. *Do each of the methods find different kinds of bugs?* and *can we use static code analysis with a hybrid fuzzer to increase the number of bugs found?*.

1.4 Research questions

This thesis's primary goal is to find bugs more efficiently by expanding on hybrid techniques, specifically hybrid fuzzing. Can we make the current tools even better? Will static analysis make them more efficient in automatic bug discovery?

We pose the following research questions to set clear goals in this thesis:

Research question 1: *What are the interesting hybrid combination of fuzzing and analysis techniques?* To answer this, we need to investigate what classes of bugs these techniques find. Then, we will empirically evaluate to see if our assumptions are valid. Doing so could indicate how we can integrate these techniques more efficiently and if there is any merit to integrating static code analysis with dynamic analysis.

Research question 2: *Can we find the same occurrences of bugs with static code analysis tools and dynamic analysis ?* We can get valuable insight into how different static and dynamic vulnerabilities analyzers detect vulnerabilities by answering this question. We can also see if there are any vulnerabilities each is better at finding. Based on the results from we get, it

support the intention and thought behind RQ1 as-well.

Research question 3: *How well do the different approaches compare in terms of efficiency and bug discovery on real world software?* After answering the first two research questions, theorizing the properties of both the techniques in questions (.i.e static analysis and dynamic analysis) and the properties of bugs, and conducting an empirical evaluation, we are left with the last question. How does this improve automated bug discovery on software?

1.5 Research objectives and scope

Answering the research questions in section 1.4 requires us to define a set of objectives for us to evaluate our proposed direction successfully. These objectives will describe how we will attempt to analyze our automated bug discovery paradigm in-depth.

1. For the first research question, we will describe a theoretical framework for bug classes. Look at the properties of a bug, how it behaves and what kind of impact it has on software. Group them in general classes, then look through our dataset containing software with already defined vulnerable bugs linking them to the classes we defined.

2. Then we will take a look at fuzzing, symbolic execution, and static code analysis technique in detail; what are their properties? How do they work? After this theoretical evaluation of the properties of bugs and analysis techniques, we will conduct experiments to evaluate our direction empirically.

3. We will also empirically evaluate each code analysis technique on the whole dataset irrespective of the bug's class and the technique's properties. This is to answer the second research question and try to verify the findings of the first research question. Comparing the results of using all analysis techniques on the entire dataset and the results of them only being used on the part of the dataset that we

deemed better suited for could indicate strategies to increase bug discovery.

This thesis aims to improve the current understanding of bug discovery using automated tools. To achieve this within the limited timeframe while still generating usable results, we need to define the scope of this research.

The techniques will also be limited to one tool, meaning fuzzing, symbolic/concolic execution, and static code analysis will be represented by AFL++[34], QSYM [73] and Facebook's *infer* [37] respectively. This may or may not impact the study results; any result will be based on the underlying structure of these tools. Meaning the tools themselves may impact the result of the experiment. For example, could another tool perform better? Are the strategies used in developing the tool a significant factor in the kind of bugs it finds? We could use multiple tools with different datasets in a more extensive study while still adhering to the theoretical framework we built to get more generalizable results. However, we will use the abovementioned tools and the dataset to limit the scope. Moreover, we will focus on **memory bugs**, which limits our findings to memory safety.

1.6 Research Method

This thesis aims to investigate if we can effectively optimize bug discovery by combining static and dynamic analysis techniques. However, we aim to use separate tools for each program analysis technique over the same data sets and then compare. The tools (i.e., *infer*, QSYM, and AFL++) are not engineered together or run in parallel (except for AFL++ and QSYM) but use their independent results to 'enhance" their own discovery. We choose this design to see if we can improve bug findings efficiently and without more design sacrifice by combining the individual tools.

Look at bug discovery time and types of bugs found by each technique individually and together to investigate the interesting combination of techniques. We choose each individual tool based on some property we

desire, for the static analysis tool:

- Able to handle large programs
- Able to analyse imperative programming languages
- Be open-source
- Present the analysis results in an easy accessible manner

By 'handle" i mean that the analysis results did not degrade too much based on size, and to do this efficiently. And since our focus in this thesis are memory based bugs we needed a static analysis tool to be able handle imperative programming languages.

The requirements we have for the dynamic analysis tools:

- Be able to focus on parts where the static analysis found a potential bug and the program as a whole
- Be fast compared to other tools
- Be open-source
- Be compatible with the languages and platforms supported by the static analyser tool.
- Be able to work with the other tools

Regarding the first bulletin, the tool must be able to analyse the vulnerable parts marked by a static analyser tools and in addition the whole source code. So we can draw some conclusions on the effectiveness of our idea. The second bulletin refers to selecting tools that have shown a focus on speed, meaning, designed to be optimized(more on this in the background chapter).

The data sets requirements:

- Contain bugs all ready found by other analysis tools
- Programming language and platform dependence compatibility with analysis tools.

- Accurate description of bugs contained in the source code
- Be open source

If the bugs contained in the data sets all ready where found by other automated analysis techniques similar too the ones we use; then our tools should in theory be able to discover them as well.

The **Success Criteria** would be to either prove or disprove our approach. Meaning, if we could see an improvement in bug findings either in number of bugs(For the LAVA data set) or when the bugs where discovered(CVE files).

The difference in how we judge the LAVA data set and CVE files are mainly that for the LAVA files [30] we will judge performance on:

- Which approach finds the first crash and LAVA bug first
- How many LAVA bugs where found

Note that, with only instrumenting part of the code it would make sense that full instrumentation of the source code probably could find more LAVA bugs overall.

The criteria for CVE files, we will only focus on detection time since we are using the existence of the CVE as ground truth variable.

1.7 Main Contributions

Defects in software are a natural byproduct of writing code. Nothing in this universe is perfect, and we are writing more and more lines of code. Code which will run in applications, on top of operating systems. Applications that will mix with other applications, running on different operating systems. Finding bugs as early as possible is optimal, but it is tough to write specific test cases. DARPA, with its competition, tried to do what humans always do; automate the process of bug discovery. Which the competitors did, mixing different software testing techniques. The result

of the CGC was a promising glimpse into how the future could look and contributed valuable learnings into the field of software testing.

This thesis's main contribution is to study how we can better integrate the different software testing techniques for an increased bug discovery rate. For example, how can static code analysis, fuzzing, and symbolic/concolic execution, coupled with a classification of bugs based on their properties increase bug discovery? Answering this will build upon the learnings from the CGC and explore the possibility of a more efficient method of validating software using automation.

The novelty of this thesis is the use of static analysis to identify where in the code a bug is and to analyze only that part and relevant other regions dynamically. Similar research has been conducted like Chen et al. [16] where the authors claim that 43% of explored code in the dynamic analysis does not contain bugs; therefore, the code coverage-centric way of hybrid fuzzing is inefficient. So instead, they propose a bug-driven approach, where the concolic execution focuses on input seeds that are more likely to uncover bugs; first, they statistically compute how far a particular seed will go and uses UBSan violations to label the region with a bug. Another is Corina et al. [19], where the authors use static ability to generate structured inputs for fuzzing kernel drivers. However, I believe no research has entirely been based on limiting the dynamic analysis and sacrificing soundness for a more effective dynamic analysis of the code.

1.8 Thesis Outline

There are five chapters, the first one being the introduction chapter.

Chapter 2: Background This chapter will dive into the relevant theoretical knowledge we need for this thesis. First, explain the DARPA's cyber grand challenge and what learnings we got from it that stimulated our research questions. Then, we will move on to the techniques used in the CGC, fuzzing and symbolic and concolic execution, and the respective tools we will use in this thesis. After that, we will dive into static code analysis

and the specific tool we will use. Lastly, software defect characterization. What that entails and how we classify defects based on their properties.

Chapter 3: Project Execution and Evaluation Her we will explain how we are going to perform our empirical evaluation. What resources we are going to use, the datasets used, and the experimental design.

Chapter 4 & 5: Results and Conclusions The last chapters, we attempt to answer our proposed research questions, go through our results and present the findings. Then, we will discuss our findings. Lastly present a summary, suggest any future work.

Chapter 2

Background

This chapter will introduce DARPA's Cyber Grand Challenge and how the competition was structured. Then, we will look into what a bug is and group the different bugs into classes based on the behavior and the impact the bug has. Following will be an introduction to fuzzing, symbolic execution, concolic execution, and static code analysis. After each technique, an in-depth look into the tools that we use in this thesis. A brief look into who the teams were and what strategies was prevalent in the competition for bug discovery and vulnerability detection(however, we will focus on bug discovery in this thesis). Finally, a framework of bugs and the observable properties.

2.1 DARPA's Cyber Grand Challenge

Hundreds of teams responded to DARPA's challenge, and after three qualifying events, seven teams remained to battle in the final round. The challenge was essentially a "capture the flag event," but with a twist, the players were machines only. The competitor's task was to develop an autonomous system that could reason about and patch vulnerabilities in real-time.

Each of the teams had the same services running on the same networked server with the same hardware. In this challenge, the competitors cyber reasoning systems(CRS) played against each other by

analyzing challenge binaries (CBs); they tried to exploit the CB to create a proof of vulnerability (PoV). A PoV could be generated in two different categories, type 1 and type 2. Type 1 vulnerability is where the competitors prove control over the instruction pointer and control a general-purpose register. A human could, in theory, craft an exploit based on a type 1 vulnerability. Type 2 is where the competition framework (CF) creates a memory region. The CRS must prove it can read a specified length of contiguous bytes in that region; before the CF judges the correctness of the data.

The CRS also attempted to patch the binary before any other competitor could exploit them. Points were gathered by either exploiting a competitor's CRS running services, patching binaries used by a CRS to the extent that others could not exploit them and keeping their services available.

DARPA developed a custom operating system for the challenge called DECREE, which is a lightweight OS open source project. Duplicability and event recording possibilities of the OS make it an excellent platform to research security in a scientific context. The Linux-like OS had only seven system calls; its architecture was an intel x86 32-bit system. There was no shared memory nor any file I/O operation. The virtual memory in DECREE allocates the pages for the stack memory automatically and makes it executable [72]. It may create an attack vector(stack buffer overflow) that the teams must consider to safeguard their binaries and system.

In the final round stood 7 teams:

- **ForAllSecure** with their CRS **Mayhem** as number one
- **TECHx** comes in at second place with **Xandra**
- **Shellphish** became lucky number three, using CRS **Mecaphish**
- **Deep Red** and CRS **rubeus** comes in at fourth place
- **CodeJitsu** captures fifth with CRS **galactica**
- **CSDS** places sixth with their CRS **jima**

- **Disekt** and **Crscpy** CRS coming in last.

When we discuss anything related to the *competitors* of the CGC, these seven teams are whom we refer to. The teams used different approaches, but every team used two main techniques: *fuzzing* and *symbolic execution*. However, they approach it differently. The differences in strategies will be explored later in the essay at section 2.8.

2.2 Fuzzing

Professor Barton Miller, in the 80s, conceived the term fuzzing; he discovered that random inputs to UNIX utilities would crash the system. He then, along with his students in his class on operating systems, created a fuzzer. They found out that 25-33% of programs crashed or hung [48] by using a simple fuzzer. The fuzzer also revealed some possible security vulnerabilities when the system crashed.

The way fuzzing works is by providing a computer program with random data as inputs, monitoring the system output or lack thereof, and examining what happens. The technique gathers information on how the program responds when it does not receive the expected inputs. Fuzzing has risen in popularity in no small part to its simplistic nature; take some input, mutate it and see what happens. There is no need to know any more than that (assuming a primitive fuzzer). Developers may use the fuzzing technique to test the quality of a piece of software, or it can be used by hackers (both white and black hat hackers) to test the security of a given software. What knowledge we gain from the crash of the program might have a bearing on the security of the said program; it might leak some information or possibly expose a vulnerability.

There are two main approaches to fuzzing [49]:

1. **Mutation-based fuzzing** also referred to as a ‘dumb" fuzzer, is when you provide a program with random inputs only. It only requires one or more templates of a correct input. It then mutates that correct input at random places. The upside with a “dumb" fuzzer is that

knowledge about the inner workings of that particular system is not needed when fuzzing. Other advantages of a mutation-based fuzzer are its ease of implementation, its portability, and ableness to find surface-level implementation errors often [71]. A drawback is its low code coverage.

2. **Generation-based fuzzing** produces inputs on its own, based on a knowledge of the system inputs structure. Unlike mutation-based fuzzing, generation-based fuzzing will have significantly more extensive code coverage, up to 76% more according to Miller and Peterson [49]. However, it requires a more intimate knowledge of the software system. Work in fuzzers has moved away from pure random fuzzers as it only would find simple programming errors. Input validation would stop most of the naive implementations of a fuzzer. The idea is to find flaws with “targeted" randomness, i.e., incorporate some heuristics and decision making instead of pure randomness, to increase efficiency and code coverage.

A fuzzer can be used either as a white, black, or gray-box testing technique. White box fuzzing is when the source code for the program and runtime information is available. The gray box is when the information is not available, but it is guided by analyzing the program responses. Black-box is fuzzing blindly, without access to the source code or analyzing the program responses to improve future mutated inputs. Black-box fuzzing is not an effective technique to be used by itself: essentially, an infinite set of test cases might be generated, most of which will not be a valid input. Most modern fuzzers in use are not a black-box testing technique but rather a grey-box to target bugs that reside deeper in the program [8]. Input mutation is usually aided by some other “intelligent" technique like static analysis, symbolic execution, or an evolutionary algorithm. Where and when the fuzzer is applied also impacts the performance, meaning to increase efficiency by actively choose when to start fuzzing and remove unnecessary overhead, like fuzzing constant steps of the program. There

might not be a need to mutate that which is constant for every execution. Competitors like ForAllSecure demonstrated this by delaying the fuzzing of a binary until user input was read in [4]. Limiting the fuzzing to certain parts might miss some bugs that could be triggered if the whole process was fuzzed. Nevertheless, like in all things in the universe, a balance must be struck.

The competitors of the CGC have all employed fuzzing in their CRS; they differ in how but all use some form of grey box fuzzer. More on that in section 2.8.

2.3 AFL: American fuzzy loop

Michał Zalewski is a white-hat hacker and a computer security expert. He worked for Google as the director for information security before moving on to snap inc as a VP for information security. Under the pseudonym lcamtuf developed a coverage-guided grey-box fuzzer known as AFL. There were two main design principles in developing AFL, speed, and reliability but also ease of use.

Coverage measurement AFL [74] calculates the coverage by injecting two byte random code at each branch point. Figure 2.1 shows how the injected value is calculated, so when a the code is executed and a new branch is reached, the current and previous location is XOR'd.

```
29
30    cur_location = <COMPILE_TIME_RANDOM>;
31    shared_mem[cur_location ^ prev_location]++;
32    prev_location = cur_location >> 1;
```

Figure 2.1: AFL edge coverage [74]

The tuples(i.e branch source and branch destination) save the directionality by doing a shift operation on the location values after they are stored, which means that going from location **A** to location **B** differs from, location **B** to location **A**(which is important because of edge coverage and not block coverage). *Shared_memory* we see referred to in line 31 in Figure 2.1 is

an bitmap where AFL calculates each byte set as a hit count for a tuple of source and destination branch.

These tuples are stored in a 64KB shared bitmap, the size is for it to fit in the L2 cache, however collisions will occur if the code base is large enough. This goes to show the design decision to speed instead of accuracy. This makes sure we can differentiate between inputs that executes the same blocks, just in a different way. AFL does that by applying a hashing function to the tuple to determine which path was taken and store the amount of time that unique path frequenzy in execution. An input that creates a new tuple(i.e new edge coverage) is prioritized while the rest are discarded, even if the new input does have a unique execution path. The developer argues that incorrect state transitions is fare more likely to trigger bug rather than the number of basic blocks traversed by an input.

This adheres to the main design principle of speed gain for precision trade-off, where edge coverage. AFL also calculates a hit-count, to calculate the frequency of a basic blocks execution. This is one of the ways AFL differs from other fuzzers used in the past, where it calculates code coverage basd on edges and hitcounts instead of basic block coverage.

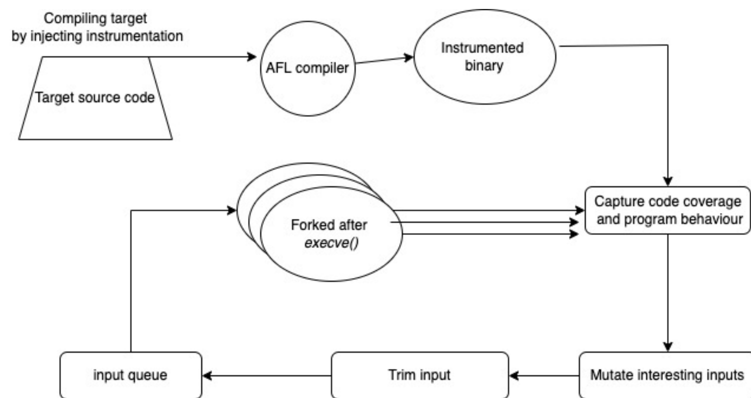

Figure 2.2: AFL workflow exploring a target program

Figure 2.2 illustrates the fuzzing process. AFL starts with a given input from the, captures its code coverage, and begins. You can also see in Figure 2.2 AFL implements a “fork server". That is to perform all necessary libraries are linked, performs *execve()*, and reaches the first

instrumented function. AFL does this to increase performance by 1.5x or 2x more efficiently instead of beginning the fuzzing process before even reaching the first instrumented function [74]. AFL also supports using a modified QEMU to gather the code coverage profile when the source code is unavailable, Figure 2.2 describes the workflow only with source.

Genetic algorithm

AFL generates inputs using a genetic algorithm to mutate inputs that are 'interesting". Interesting inputs being those that trigger a unique coverage, inputs are still randomly created but guided using a fitness function based on its code coverage. The inputs are also trimmed as much as it can be without changing code path it generated post-trimming.

2.3.1 AFLplusplus

AFL has been built upon over the years; multiple improvements and specialization of the popular tools have been engineered since its conception [18, 35, 67]. The incremental nature of improvements in fuzzing tools inspired AFL++ [34], an open-source community-driven fuzzing tool that incorporates state-of-the-art fuzzing improvements. Over time other non-AFL fuzzing research was incorporated into AFL++, even adding novel features striving to make AFL++ a state-of-the-art fuzzing tool.

By using the community-driven approach of incorporating new novel ideas into one fuzzer, AFL++ updated the comparison by using LAF-intel [38]. LAF-intel splits up large comparisons into smaller chunks, getting feedback for each chunk and evolving the mutation based on that. For example, instead of comparing an unsigned integer with a bit width of 32-bit, it splits it up into chunks of 8-bit unsigned integer comparisons.

Another technique is to use AFLsmarts [56] to use the input structure in the fuzzing campaign. The structure is guessed from the initial inputs and falls back to raw fuzzing if the coverage is not improved. This means it can fuzz an input's structure rather than the raw bytes if coverage is increased.

Other than using AFL++ for its speed, coverage, and mutation, which are superior to AFL. We choose AFL++ because of its ability to partially

instrument a binary. This way, we can focus the fuzzer only on the part of the code where we suspect there is a bug and not waste time elsewhere. Also, AFL++ uses an evolutionary algorithm that evolves an input based on coverage. This can be both a blessing and a curse. A curse is if the code being fuzzed can handle different file types, and the file types have different structures; if AFL++ evolves an input that does increase the coverage but mutates the inputs magic bytes and not the structure, it can lead to poor performance.

2.4 Symbolic execution

Symbolic execution is a program analysis technique proposed by King [40]. This technique allows to explore all execution paths in a given program. The input values are represented as symbolic values, meaning that the variables are not evaluated for their actual value but instead replaced by a symbolic expression which results in a path condition for a given executable path. A path condition is a boolean first-order logic of the symbolic expressions in that path.

A path condition is solved by a constraint solver that produces a set of concrete inputs satisfying the constraints of the program's execution path. If the end state is reachable given some input, then the path is feasible; the path is infeasible if there are no inputs that can produce the execution path. This makes it a powerful tool to explicitly test that certain conditions will not occur for any given input. A symbolic execution tree is a way to illustrate any feasible path; it is an excellent way to illustrate the idea behind the technique.

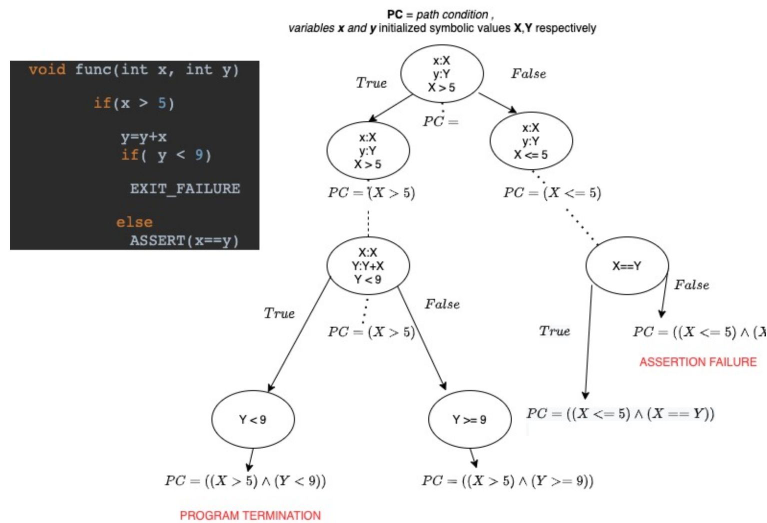

Figure 2.3: Symbolic execution tree

Figure 2.3 depicts a simple program expressed as a symbolic expression tree. The top of the tree shows the first condition. The symbolic state has a path condition (PC) that is empty; it will be populated with different constraints along the execution path, all of which must be satisfiable. The function $void\ func$ starts with an $if - else$ condition; then two branches will be created, the execution tree will follow both paths, the true branch, and the false branch.

The PC is also updated with every constraint in the specific path, meaning the PC in the true branch has $(X > 5)$ while the false branch PC consists of $(X <= 5)$. Following the true branch deeper, we can see that symbolic values stores the operation of adding the input variables $y + x$ symbolically as $Y : Y + X$ but do not compute them. At every leaf node of the execution, the tree will create a formula that can generate a set of test cases that will guarantee that the program will follow the same executable path if it is feasible.

The symbolic execution technique is a handy tool to verify that a given execution path does not occur for any given input without ever having to do a concrete execution (meaning execute the program with actual data). Its completeness [7] can be a double edge sword.

There are some drawbacks to a classical symbolic execution. First,

it has an exponential growth rate; even small programs might be too computationally expensive to be worth it. The analysis needs to parse all variables as symbolic values. If the program uses code that calls functions outside of the control of the symbolic execution and alters some conditions, then a later evaluation will fail. Furthermore, loops termination dependent on a symbolic variable can create an infinite set of paths [13], and non-linear arithmetic that the constraint solver cannot handle are some of the limitations of a classical symbolic execution.

2.4.1 Concolic testing

Concolic testing [36] is a way of dealing with most of the abovementioned issues. Instead of only handling the variables only as symbolic throughout the execution, we complement symbolic execution with a concrete execution. The concolic execution maps all the variables as concrete data; it receives a random set of inputs at the beginning of execution. Both the symbolic state and a concrete state are preserved; to force the execution away where the symbolic values may deem a branch condition solvable when in fact, it is not; it replaces the symbolic variables with concrete variables by backtracking. Imagine a constraint that is theoretically solvable but computationally infeasible. Such as finding the same hash for two separate variables; in concolic testing, we can backtrack and negate a given branch condition and replace the symbolic values with concrete ones, the branch is not explored, and the path condition is updated.

Forcing the execution away from that given branch means it can return a false negative, i.e., the program may contain an error, but finding an input that satisfies the constraints of the branch cannot be solved in a reasonable time. It means that we will lose completeness, although trading a false positive with a false negative is suitable for saving resources.

Symbolic state explosion affects both classical symbolic execution and concolic testing. An increase in conditional statements will exponentially affect the number of alternative paths a program can take. The exhaustive search approach applied by the technique will force an evaluation of all the

feasible paths. Concolic testing is better than the classical way; however, it still does not scale well. There are ways to deal with this issue, like using heuristics when choosing a path or ignoring frequently used paths.

2.5 QSYM

QSYM [73] is a practical concolic execution engine designed to be hybridized with a fuzzer. QSYM alleviated some known bottlenecks in concolic execution and optimized with fuzzing to outperform conventional concolic execution where these bottlenecks make it too computationally expensive.

The design choices made by the developers where so QSYM could scale to real world programs.

2.5.1 Bottlenecks in concolic execution addressed by QSYM

Slow symbolic emulation Symbolic emulation could be slow because of different things like path explosion or constraint solving. However, it is the opinion of Yun et al. [73] that intermediate representation(IR) is to blame for much of the overhead. Traditional symbolic execution tools adopt an IR to simplify symbolic modeling, but it incurs significant overhead. This is because the number of instructions after translating increases dramatically. In addition, the basic block (sequential instructions, with an entry and exit) contains instructions that do not need to be represented symbolically, resulting in unnecessary modeling of information not related to the gathering of constraints. Meaning, useless information also symbolically leads to more overhead.

Symbolic and non-symbolic instructions and translating the instruction in itself also cause overhead.

The optimization made in QSYM was to remove IR and use what they call Instruction-level symbolic execution. Of course, constraints were gathered directly from the instructions, making the implementations harder and limiting QSYM to one architecture(x86). However, this way

freed them to only symbolically emulate instructions needed to create symbolic constraints instead of emulating every instruction in a basic block. Tools like angr and S2E do block-level emulation, meaning all the instructions in a basic block are symbolically emulated, even if most of the instructions are unrelated in gathering constraints. Unfortunately, this would force them to rely on snapshots because overhead incurred emulating, unlike QSYM, which can re-execute efficiently.

Snapshot Usually, when a concolic execution takes paths at a conditional branch, it uses the snapshot as a starting point when the path is adequately explored. This is to save the cost of having to re-execute the entire program. Unfortunately, this does not work for a hybrid concolic execution that relies on a fuzzer for its input.

When concolic execution saves the symbolic state(where all concrete variables are mapped to symbolic ones), it is based on the input it got from the fuzzer. When the concolic engine decides to abandon the current execution and then loads the snapshot, the following input might not lead to the same path as the previous test case. This is because of the nature of how a fuzzer generates test cases(i.e., bit-flipping, mutations); they make random(albeit minimal) changes to a seed input. Also, the snapshot cannot just save the program state, recover from a snapshot and then move to a different path. External variables, such as kernel state(register values, file descriptors, system calls), can alter the symbolic status that needs to be considered. Unfortunately, the solutions to saving the external environment are expensive and increase overhead but have completeness and soundness. QSYM's approach is to re-execute as compared to recovering from a snapshot. The result is less execution overhead when using the external environment that a concolic execution engine has. However, a precision loss is to be expected, but this is where the hybrid nature of QSYM comes in because it QSYM will result in unsound test cases with no new code coverage [73]; which fuzzing will alleviate by eliminating test cases with no value to increased code coverage.

Soundness Soundness is for current concolic execution are a potential

overhead as well. While it guarantees that an input will result in execution path, if it can satisfy the constraints, strict soundness makes concolic execution fail or time-out against logic that cannot be solved by the constraints solver in a reasonable time [5].

Therefore, the designers of QSYM sacrifice strict soundness in pursuit of better performance. Optimistic solving refers to it where QSYM only solves a portion of the constraints to generate test cases, then hands them over to the fuzzer to check if it is useful quickly. Another optimization is basic block pruning, where the code repeatedly generates the same constraints. Rarely does it lead to increased code coverage, so QSYM selects the most frequent blocks and prunes those blocks. QSYM will no longer generate constraints from these blocks unless the frequency is under the power of two. Frequency derives its count from multiple executions, where the basic block is seen as a group, meaning it has to execute N times for it to be counted as one execution. For example, the basic block is **A**, and it belongs to group **G**, **G** has an execution limit N=10. Now **G** has to be executed ten times before **A** has a frequency count of 1. This limitation is built in to avoid excessive pruning blocks that could lead to interesting paths. Context sensitivity is another heuristic, where basic blocks are evaluated differently, and frequency is counted separately regarding its context. QSYM uses the call stack to handle the difference.

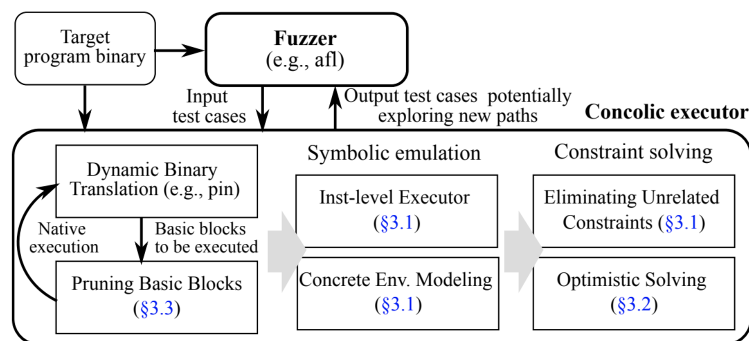

Figure 2.4: QSYM-architecture [73]

Figure 2.4 Depicts QSYM architecture; it shows an overview of how QSYM handles analyzing a target binary. Here we can see that the dynamic

binary translation is implemented using intel's binary instrumentation tool. Unfortunately, the version of PIN QSYM depends on is 2.14, which is quite old. It is not compatible with newer versions of any Linux kernel equal to or higher than 4.0, and intel made significant changes for versions 2 to 3. This means QSYM works best for Linux kernel version 3.13 and is not guaranteed to produce good results for versions 4.4 or higher.

2.5.2 Satisfiability modulo theories

Qsym relies on Z3 [50] to solve the the feasibility of a certain path; Z3 is based on Satisfiability modulo theory(SMT). It decides the satisfiability of the contstraint on said path. You can view a SMT solvers core as a SAT solver + theory solver. SAT solvers are NP-complete, so we use a theory solver to try to intelligently brute-fore a given problem. Figure 2.5 shows a high-level view of how a SMT solver works.

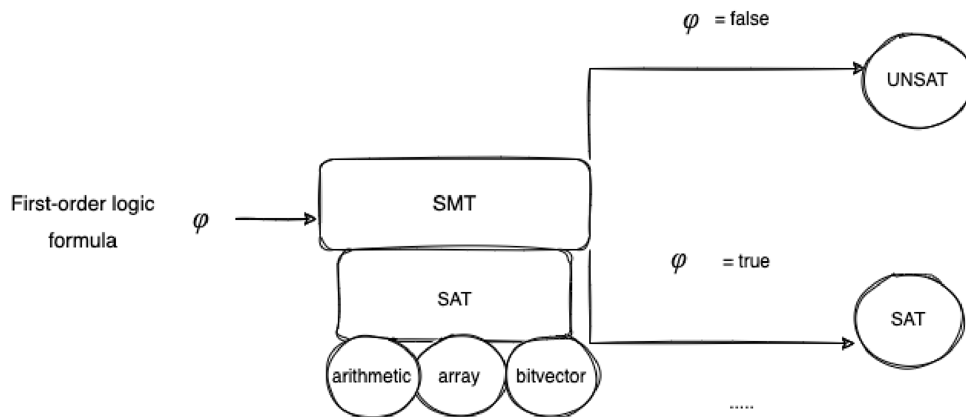

Figure 2.5: Overview of SMT solver architecture

SAT solver starts with a chain of ***N*** number of boolean variables with either of the following connectives $\neg$, $\land$, and $\lor$. It returns true if each variable *true/false* value can return true for the entire chain; if not, it returns that it is unsatisfiable. The chain can be based on many theories, and some supported theories that Z3 supports are arrays theory, bit-vectors, lists, arithmetic, partial orders, and tuples. Although the most used form of SAT solver is to use the conjunctive normal form(CNF), there are other forms

as well, such as the disjunctive normal form(DNF) and negation normal form(NNF). CNF is the most popular because it can take a boolean formula and convert it into CNF with polynomial size, meaning the formula can be simplified while still having it in a reasonable size.

2.6 Static code analysis

Static code analysis evaluates computer software without actually executing or running the code. The technique checks for the correctness of the program used but can also be applied to optimizing the code and trigger warnings when compiling source code. A famous issue for static analysis is that the technique suffers from a high number of false positives since it essentially is 'guesswork " [6]. There are multiple paradigms for performing static code analysis.

Control analysis is one of those paradigms of static code analysis. This analysis refers to the representation of a script or a piece of code as a graph in which the graph nodes represent the basic blocks of the program, and the edges connecting the nodes represent the paths and connections that exist between the graph nodes. It might happen that a node only has an exit edge (a path to a different node from that node); in this case, this node is known as the entry block. Similarly, if a node only has an entry edge (a path to that node from a node), it is known as an exit block.

Data analysis Data analysis in static code analysis refers to the method of data mining and learning relevant available codes to infer coding rules. These inferred coding rules can then be used to create automated software that takes the program to be analyzed as input and apply the inferred coding rules to ensure that the program satisfies these criteria. Also, the same method can be used to study past fixes that also help with revising and correcting the issues found [42]. Also, data flow analysis can be used in static software analysis in which run time data is collected about a program in a static state by utilizing symbolic values.

Interface analysis is where the program is checked by a software that

verifies interactive and distributive simulations. A subcategory of interface analysis is user interface analysis analyses user interface model, that is the elements and definitions that related to the interface defined in the program code. This analysis tries to ensure that there are no problems in the interaction of the user with the program. It is necessary to mention that the model is also looked at as a whole and in the context of the entire program to ensure that the relevant interactions of the interface with the remainder of the program are also without vulnerabilities.

Most (if not all) static analyzers have a high false positive rate because of trying to make the tool sound. This is because of Rice's theorem, which means that in software security analysis, we cannot guarantee that a piece of code is bug-free, and a high false positive rate is far better than a high or even moderate false negative rate. Yet we aim to alleviate this high rate of false positive by combining static and dynamic analysis.

2.7 Facebook's *infer*

In September 2013, Facebook acquired a company called Monoidics; they were adapting academic research into usable industry program analysis tools. The first iteration of *infer* was verifying memory safety in C code [14]. Later it was developed further at Facebook to handle various bugs and issues. Such as buffer overflows, integer under/overflow, dead stores, resource leaks, data races, and more. *Infer* is open source, coded in OCaml, and can analyze C/C++, Objective C, and Java code.

A high-level description is that *infer* analyses code in two phases, *capture* and *analysis* phase. In Figure 2.6 the front end is where the compilation commands for the target source code and builds the system. Then, it translates into an intermediate language called Smallfoot Intermediate Language (SIL); this is the capture phase. The backend in 2.6 is where the second phase begins, *infer* takes in the SIL and analyzes it for any bugs.

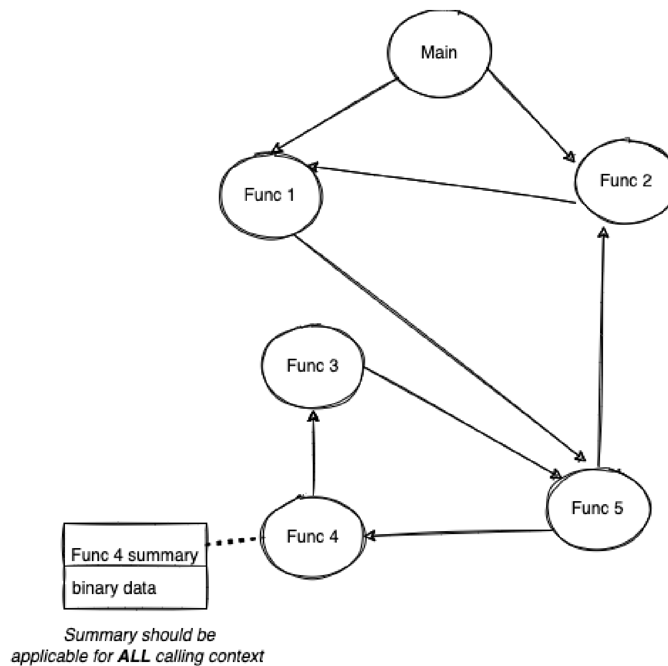

Figure 2.7: Infer call graph [17]

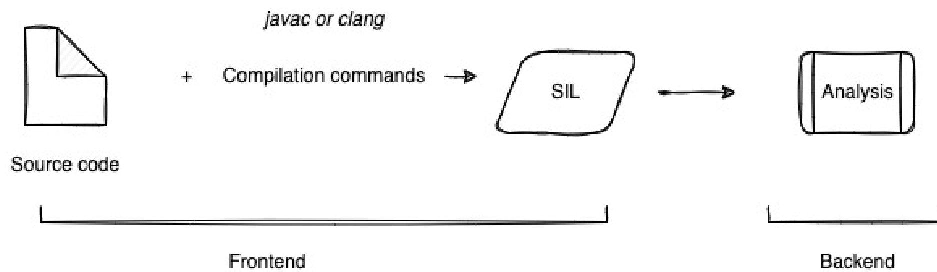

Figure 2.6: High level view of infer's workflow [17]

Facebook deployed *infer* in their CI/CD pipeline, where in every change made by a programmer, an instance of *infer* was deployed as well. This made the requirements of *infer* strict; to analyze small changes while still depending on the program context as a whole without re-analyzing, i.e., if procedure **A** was changed, then only **A** and its direct dependencies were to be analyzed again. They solved this issue by doing compositional and incremental analysis on function summaries; this way, they could achieve inter-procedural analysis; this is what happens in the backend phase of 2.6. Figure 2.7 illustrates how *infer* handles inter-procedural

analysis. It generates a call graph, the graph is directed to illustrate dependencies, then it evaluates each procedure/function starting from the leaf nodes. So, it starts from *Func 4* and generates a function summary; and since *Func 4* depends on *Func 3*, it either uses a function summary or generates one. Then it uses *Func 3* summary and evaluates *Fun 4* and *Func 5*. Some functions may also be run concurrently, which *infer* checks for. This leads to parallelization.

The inter-procedural analysis may introduce cycles, which *infer* must resolve before the analysis can terminate. Let us say *Func 4* needs the function summary for *Func 5*; the analysis of *Func 4* is then paused; however, *Func 4* needs *Func 3* summary, and now both *Func 4* and *Func 5* are paused. Now *Func 3* needs *Func 4* function summary, and with that a cycle has occurred.*Infer* then uses a fixed point to break that cycle. *Infer* is non-deterministic due to parallelism; a random selection of starting node *infer* begins its analysis on and cycles.

2.7.1 Separation logic and bi-abduction

To reason about buggy code, the developers of *infer* choose separation logic [54] for its pointer analysis capabilities, reason about computer memory mutation and bi-abduction [15] to reason about data structures. This formal verification and compositional analysis of the source code allows us to reason about bounded correctness even in large programs with thousands of lines of code. Although we are also dealing with imperative programming, Hoare logic can handle this type of programming where the programmer handles the control flow.

Separation logic builds upon Hoare logic, which has a set of logical rules to reason and analyze computer programs' correctness. Its main feature is the *Hoare triple*, where where there are a pre and post-condition separated by a command. It takes the form: *{P}* C *{Q}* where *{P}* is the pre-condition, *{Q}* is the post-condition, and C is the command or program code. The pre and post-conditions are regarded as assertions, essentially a statement made about a true program before and after the program

is executed, the command C, and terminated. With that, we get partial correctness. Total correctness is when the pre-condition assertion is true, and after the command C, the program must terminate, and post-condition C also holds true.

For separation logic, using Hoare triplets, the logic is based on a connective $*$, which can be read as *"and separately"*. So that *{P}* $*$ *{Q}* means that *{P}* and *{Q}* disjoint parts of the same memory; as such, the proofs built using separation logic are much shorter than other techniques that reason about correctness, especially when data structures are mutated because of the separating conjunctions [57]. Since there are no 'sharing" between disjoint parts *{P}* and *{Q}* when there is an update to *{P}*. This local specification can be generalized and extended to any memory, the rule known as the frame rule:

$$\frac{\{P\}C\{Q\}}{\{P\} * \{R\}C\{Q\} * \{R\}}$$

[57]

Where *{R}* is the frame, and if the program execution satifyes *{P}* then same can be said for *{P}* $*$ *{R}*. Meaning, we can perform local reasoning using the memory accessed while separately use the frame to describe memory outside the footprint.

Infer uses an internal theorem prover to find and reason about the pre-and-post condition; however, they use **bi-abduction** to find missing states needed by the theorem prover and still be computationally feasible. Bi-abduction [15] is where *infer* derives logical conclusions for separation logic pre/post condition directly from the code. It is a form of abductive reasoning using the frame rule, more specifically inverse of the frame rule. This mix allows an automated way of gathering information about the code without manually writing pre/post conditions, allowing the scalability of *infer's* bottom-up analysis. Even with a high number of incremental changes to the code, the analysis is still feasible.

2.8 Tactics employed in the CGC

Even though the ranking in the competition was clear, it does not mean that team that placed first was exceptionally better than the last-place team. This is because DARPA used a relatively complex scoring system to decide points given and taken away, which took in uptime, how many vulnerabilities they exploited in contestants' systems, and more. This section will explore how some of the competitors found vulnerabilities in the challenge binaries (CB). Examining the design designs, they made to increase performance.

2.8.1 Bug discovery

Fuzzing tools are one of two main techniques heavily used by the competitors to discover bugs in a given CB; most of the competitors used an "off the shelf" fuzzer but made some adjustments. American Fuzzy Lop (AFL) was a popular choice; Shellphish, ForAllSecure, and TECHx applied AFL as a basis for their fuzzer; all three teams ranked at the top of the competition. The second technique was symbolic execution, mainly custom-built by the competitors.

ForAllSecure used AFL as a basis for their fuzzing but went another way to calculate the code coverage. They patched the binary with their full-functioning rewriter (FFR); the FFR inserts a patch into the middle of the binary. Of course, writing into the middle of the binary required them to fix the now out-of-place addresses and offsets [4]. Rewriting a binary does require much work, but it gave them an option to rewrite some instructions to their benefit. Breaking apart 32-bit comparisons to 8-bit and fuzzing them separately and delayed fuzzing until the user input was read in gave them a 100-900% performance boost as opposed to AFLs QEMU mode according to Avgerinos et al. [4].

ForAllSecures also utilized symbolic execution, more specifically concolic execution, to explore different paths for a given input that might trigger a bug in the binary. Not only did they do the standard negation of

branch conditionals, but they also modified memory pointers to find if the modified memory accesses create an input that explores a new path. They repeat the execution with all the different inputs their symbolic execution engine generates; this will inevitably create exponential paths to explore. To alleviate the “path explosion" problem, they used the technique Veritesting [3]. Veritesting technique is a way to increase code coverage while still dealing with the “path explosion" problem. It does so by merging paths by alternating between dynamic symbolic execution (concolic execution) and static symbolic execution. ForAllSecure tactic was to combine symbolic execution with their modified AFL fuzzer and run them in parallel; the fuzzing engine shared promising inputs with the symbolic execution engine and vice versa.

Shellphish Shellphish used a fuzzer known as Driller [65]. Driller is not a pure fuzzer; it exercises selective concolic execution to make it more able to pass complex inputs. Its selectiveness is that it only applies concolic execution on inputs the fuzzer deems interesting.

Driller's fuzzing capabilities build upon a version of AFL. They altered the QEMU emulation in AFL [60] by keeping the random seed consistent for every execution until an input produces a crash; then, the concolic execution steps in to handle any “Challenge-response" by looking for that specific random string. Fuzzing for new execution paths is now possible since the randomness behavior of the program can be kept consistent for further exploration or proving a vulnerability. Keeping random seed values consistent is similar to TECHx's non-determinism strategy; both strategies try to fuzz a minimal part of the process. All this allowed them to move the AFLs fork server logic until the first call to receive. Similar to the ForAllSecure team tactic of waiting for user input before implementing fuzzing.

Angr, the symbolic execution engine used by Driller, assists the fuzzer by finding new execution paths the fuzzer cannot reach. Meaning, if Driller meets a branch condition, and the fuzzer cannot explore both paths, it passes it over to angr. Angr then uses pre-constraining to match the

execution path the input used by the fuzzer created, then tries to solve the condition fuzzer could not solve, it then executes without the pre-constraints to find a new possible path. If it succeeds, it tries to produce input(s) with its constraints solver and passes it back to the fuzzer.

TECHx [53] made several modifications to the AFL fuzzer, as ForAllSecure and Shellphish did. Some of the adaptations to AFL were to remove non-determinism by replacing calls to the random number generator with fixed values. Transmit (UNIX equivalent write) system calls were ignored, and receive (UNIX equivalent read) system calls was replaced with a no-operation instruction when it returned zero bytes. TECHx also moved the AFLs fork server right before the first call to receive, sacrificing some precision for performance, which shares a similarity to ForAllSecure's postponement of fuzzing until the program reads in user input.

Like ForAllSecure's approach of using FFR, TECHx also used binary instrumentation to gather code coverage. However, ForAllSecures begins with their custom FFR and defaults to emulation with QEMU, while TECHx runs some instances QEMU based and some instances using a binary instrumentation tool in parallel. Like Shellphish, their fuzzer handles conditionals needing a specific input where a fuzzer might have some trouble using symbolic execution; TECHx custom-built a symbolic execution engine called Grace2. They differ when they use symbolic execution; TECHx begins with assisting their fuzzer with symbolic execution to find inputs that generate new paths unlike Shellphish more conservative approach of waiting.

2.8.2 From Bug To Vulnerability

The competitors also needed to turn a bug into a vulnerability after fuzzing the CB's and executing them symbolically. Input that crashes a program can be a design flaw and not necessarily a security flaw. Each of the teams again leverages either symbolic execution or fuzzing to generate a PoV.

ForAllSecure feeds the input that crashed or exposed a bug in the binary, along with the binary itself, into an automatic exploit generation

(AEG). The crashing input is mutated to seek control over the EIP (type 1 vulnerability) or leak data from the CF specified memory range (type 2 vulnerability). With the given constraints, it symbolically executes the path and tries to develop an exploit by generating a formula; if the constraints solver deems the formula satisfiable, it can then prove a vulnerability.

TECHx has split this issue into two approaches, Quick Exploit Finder (QEF) and Symbolic Exploit Finder (SEF). The QEF tried to exploit type 1 vulnerabilities by examining the relationship between register values and input when crashing. A code injection attack was used when control over the instruction pointer was proven and not a general-purpose register. Type 2 POVs were explored using the output of the fuzzing during bug discovery; QEF also tries to modify the contents of the memory range to determine if it can generate a valid type 2 POV. QEF was fast and simple but highly dependent on the crashing input. The remedy to QEF sensitivity was the SEF; SEF used the crashing inputs to determine the scope of control over the instruction pointer and registers related to the crash it has. When it has sufficient control, it tries to generate a set of constraints that must be solvable to craft a valid type 1 POV. TECHx found out during the competition that the QEF was enough to craft POVs, even with its sensitivity to changes to the crashing input.

Shellphish strategy was to use a fuzzer they named PovFuzzer. It takes a crashing input and mutated one byte at a time while keeping track of the register values; this is then repeated many times over. One of the PovFuzzer limitations is not handling complex inputs well. Rex, the symbolic execution engine, addressed that limitation. Rex symbolically traces the execution path of the crashing input. After the crash, it uses a constraint solver to generate a valid POV or find a set of inputs that allows the continuation of execution. The team also symbolically traced ‘ìnteresting" inputs discovered by Driller that do no produce a crash. They coined this technique Colorguard. After the trace, it evaluates if it can produce a valid type 2 POV using symbolic formulas.

2.9 Software defect characterization

In this section we are going to define what a bug or software defect is. How a bug is understood in the context of bug discovery and why we choose to classify bugs into categories

> *"A software bug is an error, flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways."* [70]

The term bug was made famous in computer science from an incident on September 9, 1947, where a moth entered a computer named Mark II and disrupted the electronics; it was one of the first recorded incidents of a bug(in the world of computers), arguably the most famous one [68]. From that day, the term bug evolved with the evolution of software and hardware. It is now a more complex entity; discovering a bug is a complex task. A bug defined as a flaw or failure can mean anything, which means any flaw or failure can be considered a bug. For example, it can be a failure to convert imperial units to metric units between components like NASA's million-dollar disaster. The part of the software that calculated the total impulse reported them in pounds-force seconds while another part that calculated trajectory expected them in newton-seconds, resulting in total mission failure [69]. Alternatively, a flaw can be benign, resulting from a misunderstanding between parties involved in software development, resulting in a working software product but not one that does the intended job.

The current way we have defined a bug, being a flaw or failure, is not good enough to accurately answer *research question 1*, nor is using the individual explanation for each bug in the dataset. Furthermore, studying the 'symptoms" of the bug is not a way to treat similar bugs. Instead, we need to categorize them based on observable properties, allowing us to group them to study the efficacy of the different automated bug discovery tools.

There are different approaches to classifying a bug class, like Cotroneo

et al. [20] did an empirical study on bug manifestation based on *trigger analysis*. A trigger is what the bug needs to manifest, what input it needs, the number of inputs, and what environment it requires. As they see it, the bug chain consists of input, environment, and then the bug. The impact of the bug chain is a failure, which means they do not consider bugs that do not cause the software to fail but still could affect the program.

2.9.1 Bug type

This section will investigate the sub-categories of the bug dimensions used to group similar bugs together. First, we will review the current literature on these different bug types to gather observable and generalizable properties of each bug type exhibit. Doing so creates an abstract way of evaluating bugs as a group; and how that group could, in theory, be discovered.

Memory bugs: In this thesis, we will define memory bugs based on Bojanova and Galhardo [11] orthogonal classification of bugs that are vulnerabilities. The authors define a *vulnerability* as a weakness type that leads to unintended use of the software/hardware, i.e., security violation. Weakness type is where different vulnerabilities can have the same or multiple underlying weaknesses. Causality is the focus of Bojanova and Galhardo [11], meaning understanding the causal relationship of the underlying weaknesses that leads to a bug. Li et al. [45] *cause-impact* criteria are similar to what is being described as *cause-consequence* in Bojanova and Galhardo [11]. Where they differ is that Bojanova and Galhardo [11] uses *cause* for the improper state(s) that leads to a failure and between the chained improper states have operations that use ill-formed data.

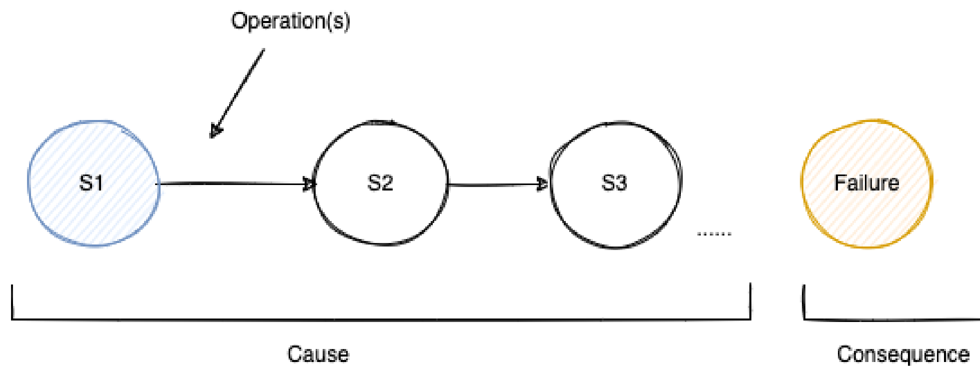

Figure 2.8: Figure describing the cause consequence classification in [10]

Meaning the first improper state, *S1* leads *S1* by transitioning because of an *operation* using ill-formed data and, eventually, failure *F*, which is the *consequence*, Figure 2.8 illustrates this.

Tracing the state transitions from failure to the first improper state might fix or find the bug and could be used to predict which analyzing technique might find that type of vulnerability or bug class more efficiently. Similar to Cotroneo et al. [20] *complexity analysis*, the authors [11] mention that for a bug to present itself, it may need several *causes*; this classifying criterion might also be an essential variable in evaluating how the static and dynamic analysis tools perform in finding the different bug classes.

The memory bug model: MAD, MAL, MUS, and MDL

Bojanova and Galhardo [11] define a *memory bugs model*, where they refer to a bug as an operation(s) over a piece of memory storing a data structure or some primitive data that a piece of memory has two attributes; a boundary and a size. Where the address has a pointer or is calculated as an offset of the stack, that is referred to as an *object*. When an object is under address formation, this phase's bugs refer to *Memory Addressing Bugs (MAD)*. In the phase where the object is allocated, the class is now *Memory Allocation Bugs (MAL)*. If the object is used under some process, *Memory Use Bugs (MUS) class*. Finally, if the object is in the deallocation phase, the class is *Memory Deallocation Bugs. (MDL)*. Under each phase(i.e., MAD, MAL, MUS, and

MDL), certain operations belong to and are separated by each phase. For example, MAD consists of four operations in that phase; initialize a pointer to an object addressee, and then there is reposition where the pointer is moved within the boundary of the object, and lastly reassign when a pointer shifts to another object addressee entirely.

The MUS phase has five operations; initialize, where it writes data in the memory space of an object for the first time. The following two phases are read and write; the former is when it reads data from within the boundary of an object, and the latter is when it writes within the boundary of an object. Clear is the operation of writing inside the boundary of an object for the final time prior to the deallocation of an object. Finally, the operation deference is when an object is inaccessible.

For MAL phase has tree operations. For example, it allocates creating space for the object, extends, increases the object boundary and size by extending the memory, and reallocates–extend similar to extend; however, it allocates a larger piece of memory and passes the object's data to the newly allocated memory space.

The MDL phase has, where it releases part of the memory, the part being the boundary and size of an object; this operation is referred to as the deallocate operation. The reduced operation is similar to deallocating. However, the difference is that the reduction in the object boundary and size is only partial; furthermore, it redefines the original object's new boundary and size. Lastly, there is the reallocate–reduce, also bearing similarity to the last two operations. Here, the distinction is that the object gets moved to a newly allocated memory space by copying the contents, moving the pointer, and deallocating the former memory space occupied by the old object.

Each cause-consequence with a set of operations comes with attributes with the memory bugs model. They give concrete context to the operations that happen over state transitions. Shortly, attributes describe the bug and its severity. Finally, these attributes describe the object and operations. The attributes describe each class in three categories; source

code, execution space, and location. These also have several subcategories further explaining the attributes; for **source code**, we have a *codebase, third-party, standard library, and language processor*. There are *user-land, kernel space, and bare metal* for **execution space**. Lastly, the **location**: *stack or the heap*. Memory addressing bugs(MAD) has one more attribute category, and memory under use bugs(MUS) has two more. Firstly, the additional category shared by MAD and MUS is referred to as a **mechanism**; it has two descriptions, *direct* or *sequential*. The former is when the operation is performed on a specific singular object, while the latter is after **N** number of objects. MUS has again one last attribute called **span**, with the descriptors *little, moderate, and huge*; it details the object's size. Memory Allocation Bugs (MAL) and memory deallocation bugs (MDL) also come with additional attributes. They are referred to as **Mechanism**, and **ownership**, where the former describes if it was an *implicit* operation(i.e., outside a function) or *explicit*(i.e., within a function); the latter describes the *ownership* of the object. The classification made in memory bugs model is an orthogonal classification, meaning there should be no overlap between the classes of memory bugs. Below we will try to showcase how we can use this memory bugs model [11] to classify a CVE. Let us apply the memory bugs model to CVE-2017-12762[22]. In the ISDN sub-system of multiple Linux kernels(4.9, 4.12, 3.18, and 4.4), a buffer overflow vulnerability existed in the ioctl handling. A user-supplied buffer is read in at isdn_common.c, and the isdn_net_newslave function call strcpy; however, the operation is performed without checking for length and the destination buffer has a fixed size. This creates a buffer overflow that could lead to denial of service or possibly arbitrary code execution.

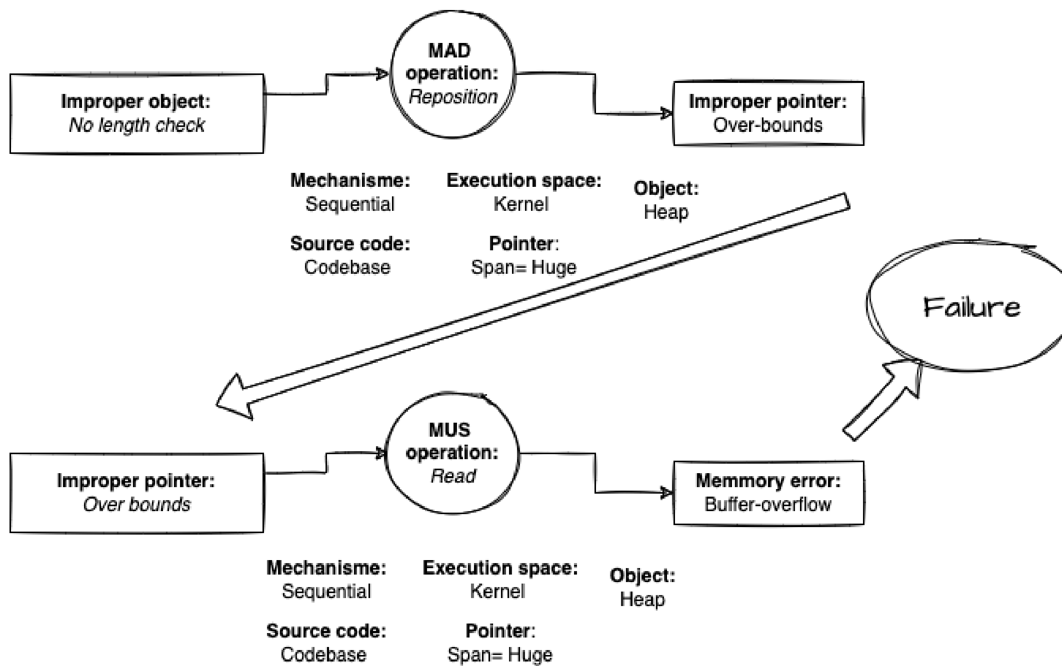

Figure 2.9: Shows the improper state transitions that leads to the software bug CVE-2017-12762 [22]

In Figure 2.9, we can see how the memory bugs model works when defining a vulnerability, where the classes of bugs encompass all the needed components from the starting state (i.e., where the program went from normal execution to improper execution) to failure. The chain of transitions begins by copying the data to a fixed-size buffer without checking the length of the user-supplied buffer; that is the *cause*. The *consequence* is reading over-bonds, ending the **MAD** phase. Now the **MUS** phase has the over bounds read as a *cause*, leading to a buffer overflow *consequence*. Each of the phases had attributes that give context to how this happens. Such as where it happens(i.e., kernel or user-land), which part of the memory, and if the bug is in the source code or somewhere else(third-party code).

2.9.2 Evaluate static and dynamic analysis techniques with regard to the memory bug model

To evaluate the techniques and their bug-finding abilities, we leverage the orthogonal classification of the memory bugs model that we looked into in detail in section 2.9.1. To find the most interesting combination of automated bug discovery techniques, knowing if a particular technique is usually better at finding a type of bug that might aid in the bug discovery efficiency by combining techniques better. For example, let us say an organization uses a static analyzer in their CI/CD pipeline and marks a program as having a *memory under use(**MUS**)* type of bug causing a buffer overflow. Then, instead of using human analysis, that possible buggy program could be fuzzed if fuzzing is more likely finds that type of bug instead of using a hybrid fuzzer, wasting more computation than could be needed elsewhere.

Later, we will go through the empirical results from Yun et al. [73] to evaluate concolic execution and classify the bugs according to our memory bugs model. For AFL++, we will go through selected trophies caught by the fuzzer and do the same. Finally, use *infer* on both QSYM's and AFL++ results. That could give us insight into how we can better integrate the different techniques. Since, according to our memory bug model, a *crash* could have a chain of state transitions with multiple bug classes, we choose to use the first bug class that appears from normal state to improper state. For example, Figure 2.10 shows a crash with two states, **SX** and **SY**, both are needed to produce a crash. However, we can avoid a crash if we resolve **SX** earlier, even without resolving **SY**. Therefore, we choose to classify each bug as the first bug class only and evaluate each tool's effectiveness based on that. Meaning the first transition from *proper* state to the *improper state*, that bug class is what we will refer to as the bug for that program.

Let us look at the program *vim* and how we will classify the bug as an example. Vim is a UNIX editor where AFL++ found a use-after-free vulnerability. The auto-command feature executes automatically when

some events occur. So, when a window is closed by auto-command, the buffer regarding the window is de-allocated. However, the pointer to the buffer is still being used. So, the *cause* of this bug is a *dangling pointer* since the proper state is changed when auto-command executes as expected. However, the freed object(i.e., the buffer) can still be used in later execution, and when it does, a crash occurs. This can be classified as an **MDL**, with the *cause* being a *dangling pointer*, the operation being *deallocation*, and the *consequence* as an *improper object for next operation*. When the pointer is used again, it transitions into a **MUS** class. *Cause* as a *dangling pointer*, operation for state transition as either read or write and finally use after free as a *consequence*. Resolving the bug when the class was an **MDL** would also resolve the later vulnerability.

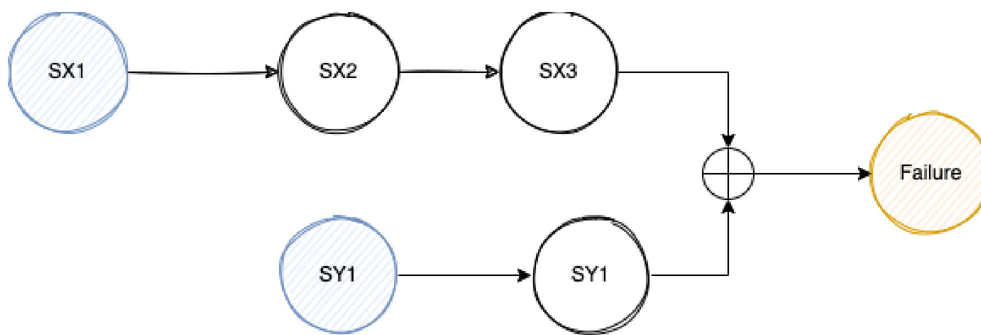

Figure 2.10: Connected state transitions [10]

Chapter 3

Project Implementation and Evaluation

3.1 Planning

There has been much research into fuzzing [8, 9, 49, 51], concolic execution [47, 59, 73], and hybrid versions of these techniques. However, evaluating the techniques on their bug-finding ability is a non-trivial task because unless the datasets have ground truth, each 'interesting" input must be manually debugged to see if it is a valid finding. Past research into fuzzing tools usually has simple evaluation metrics such as unique crashes; that do not speak to security vulnerability but more so to fault tolerance; furthermore, it is not expressive enough to reason about the type of bugs found and why. They may use automated exploration/exploitation tools; however, that shifts focus on these tools and their soundness in their reporting.

Using a heterogeneous dataset with many bug classes in real-world software, both bugs deep in the program and surface-level trivial bugs. That, coupled with ground truth to verify if the bug discovery tools found is a bug, is a much better evaluation metric of the technique's ability to find bugs. Unfortunately, such datasets are rare and hard to produce; there are some out there, like the LAVA dataset or CGC challenges binaries [53, 60,

Datasets		
Code	CVE/LAVA	bug
file < v.5.32	CVE-2017-1000249	Stack-based buffer overflow
yara.3.5.0	CVE-2017-9438/CVE-2017-9304	stack consumption
lepton.1.2.1	CVE-2017-8891	DoS
gifdiff	N/A	null deference
openjpeg	CVE-2017-14151	heap overflow
jpegs	LAVA	125 bugs
yamlB4	LAVA	5 bugs
tinyexprB2	LAVA	4 bugs

Table 3.1: Table with number of files tested, their CVE or how many synthetic bugs within the source

63]; however, they have weaknesses with the datasets mentioned above that must be accounted for when evaluating using the techniques on them. LAVA's synthetic nature might not be comparable with real-world bugs [12], and CGC challenges are modeled after real-world bugs; however, it is still not naturally occurring bugs.

To try to get as accurate results as possible, we will evaluate our empirical research on two datasets:

- Selected challenges from the LAVA dataset[1]
- Selected CVE-fixes that other bug discovery tools have evaluated [34, 73]

We will exclude the CGC challenges since *Infer* is incompatible with the source code. In so doing, we can draw some conclusions about the performance of each technique to find bugs alone and when they are used

[1] https://rode0day.mit.edu/archive 18.09, 19.07, and 19.03

together. Table 3.1 describes the software, software version, CVE or LAVA tag, and vulnerability description.

3.2 Ex3

For the experimental evaluation done in this thesis, we will use the Simula Research Laboratory computing cluster called Ex3[32]. The Norwegian research council funds the infrastructure. The Ex3 offers a heterogeneous computing cluster; we will use AMD EPYC [31] 7601 32-core Processor; the node has 64 cores, with 2TB ECC DRAM and 4TB NVMe DRAM. However, we will not use all cores or memory in our evaluation as we are on a shared resource.

3.2.1 Singularity

AFL, QSYM, and *infer* are tools with different dependencies and system requirements. Therefore, running these tools on the same platform requires us to employ a containerized application of the tools, not only for the system dependencies but to circumvent the need to have root access.

Singularity [62] offers such virtualization, where we can install and use the dependencies needed yet also bypass the need for root privileges a tool might need to be installed. We build the container image remotely and deploy the built image on the cluster for our empirical evaluation. Since root privileges are needed to build an image, we made use of a remote builder [2].

To build a singularity container, you need a definition file to specify the base OS, dependencies, how to start the container, environment variables, etc [62]. The definition file we used for AFL++ and QSYM was taken from the GitHub repository with multiple fuzzing definition files [3].

[2] https://cloud.sylabs.io/

[3] https://github.com/shub-fuzz

3.3 Datasets

This section will present the datasets we will use in our empirical evaluation. Analyze and present the bug and its characteristics and classify them according to our memory bug model defined in section 2.9.1.

3.3.1 CVE dataset

One of the real-world programs that we use in our datasets for the empirical analysis is **file** [21], which contains a stack-based buffer overflow. The program in question is the Unix file command that scans a data of an input *file* to guess the file format. After a commit in 2016, a bug was introduced where it became possible to produce a stack-based buffer overflow. For example, an attacker could exploit 20 bytes on the stack by creating a specially crafted *.notes section* in the ELF binary. This bug was possible because of a condition that is always true. We classify this as a **MUS** class using our bugs model. Meaning the *cause* of the transition into an improper state is an erroneous check on the length of data being copied into a fixed-size buffer. So then, according to our memory bugs model, the *cause* being an **memory error** erroneous *write* to object; *consequence* being a buffer overflow. We can see the attributes needed to expose this bug in Figure 3.1.

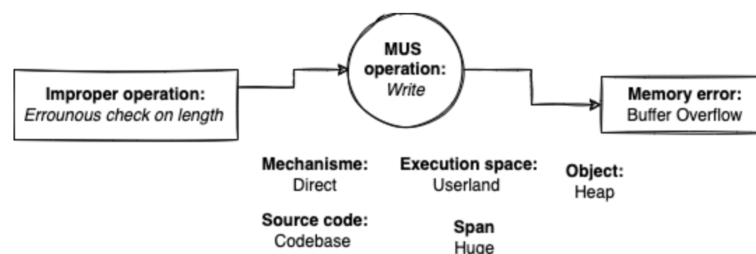

Figure 3.1: file CVE bug class [11]

The following sample in the CVE dataset is **yara**[25], which is a helper tool for, but not limited to, security research. It performs pattern matching in text and binary files based on crafted rules. The code has a regular expression module with a bug triggered by the mishandling of hex strings.

The mishandling grabs too much of the stack resulting in a crash. There was no limit on the depth of the abstract syntax tree representing the hex string. Therefore, we can classify this bug as a **MAL** class, with *cause* no limit on the amount of space on the stack the program is allowed to grab. Figure 3.2 shows the attributes and the single-state transition. The state transition from proper execution into when the program *erroneously* allocates memory, the *consequence* being a memory overflow.

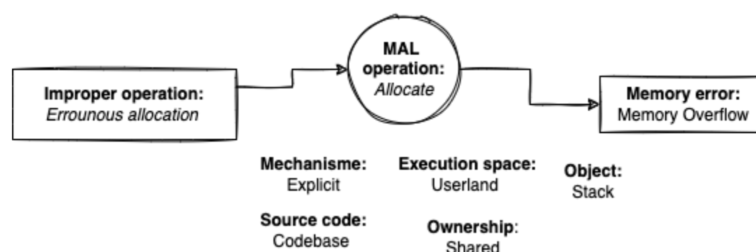

Figure 3.2: yara CVE bug class [11]

Lepton [24] is part of Dropbox; the code performs compressions losslessly on images. The application crashes by not setting up the correct number of threads if a malformed file is passed to the program. The number of threads might be more or less than the actual number of threads used; however, the buggy version still iterates over N number of threads and performs operations over their memory space. The *cause* was an erroneous *operation* because the bug was caused by an implementation error causing a *reposition* of a pointer to a wrong position. The *consequence* is over/under bounds positions of the pointer. At this point, the bug is a **MAD** class bug. Later in the execution, when the pointer is used to read some data, is when the state transitions from a **MAD** to a **MUS**. The cause of this transition is because improper pointer that is over/-under bound. When the *read* operation is performed, causing a crash/failure is the *consequence*. While still following the reasoning we defined in Section 2.9.2, we will consider the CVE as a **MAD** class bug, and Figure 3.3 shows the attributes of the CVE.

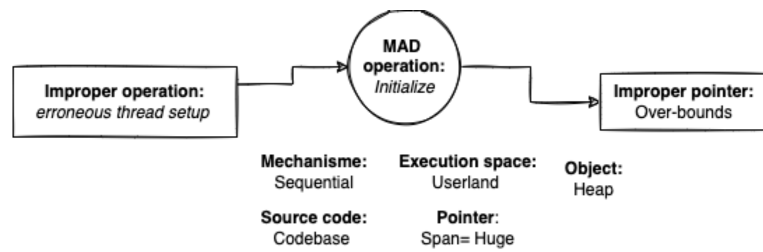

Figure 3.3: Lepton CVE bug class [11]

Gifsicle [41] is a application for manipulating GIF images. It can merge GIFs, extract component frames, and more. The code has a null deference bug caused by iterating over possibly non-initialized pointers when setting up the color map, causing the application to crash. If we apply our memory bugs model to this bug, it is a **MUS** class. The *cause* is an improper pointer that reads over bounds, resulting in a *consequence* being a pointer dereferenced. The proper state and improper state only have one state transition. After that, the execution is normal until the pointer is dereferenced. Figure 3.4 illustrates the *cause-consequence* and all attributes, which shows that it is not a relatively deep hidden bug.

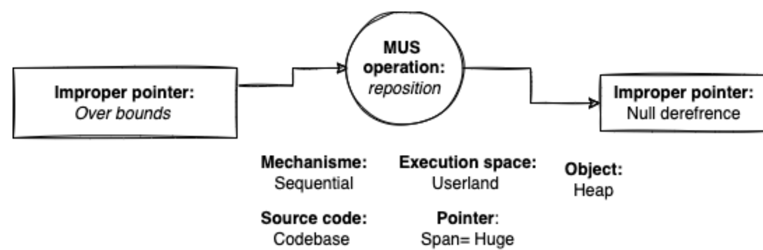

Figure 3.4: Gifsicle CVE bug class [11]

Lastly, the final target in the CVE dataset is **openjpeg**. openjpeg is an image compression library for encoding and decoding images. The code contains an of-by-one error that triggers a heap overflow. When the program compresses a file and tries to encode blocks, there is a possibility of allocating memory, causing an off-by-one error later. This is possible because in opj_mqc_byteout() in lib/openjp2/mqc.c points to before the buffer. However, this is not accounted for when setting the buffer in opj_tcd_code_block_enc_allocate_data(). With that, the *cause* here

transitions from proper state to improper state is an erroneous allocation, where the *consequence* is not enough space allocated, a memory allocation bug(**MAL**). However, the **MAL** bug transitions into a **MAD** class by *cause* being an improper object. The *consequence* is an over-bounds pointer by repositioning the pointer. The result is that this bug is both a **MAL** and a **MAD** class.

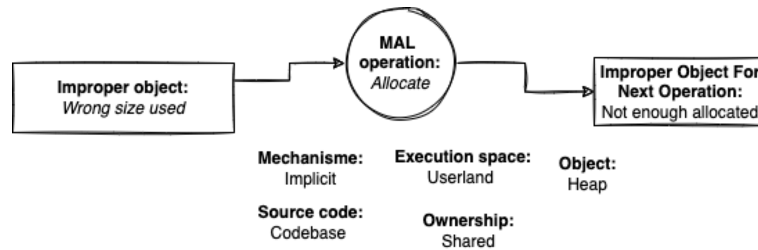

Figure 3.5: Openjpeg CVE bug class [11]

However, we classify this bug as a **MAL** class, as shown in Figure 3.5; fixing this first would fix the entire bug. Therefore, it would be more valuable for a bug discovery tool to expose this first.

3.3.2 LAVA dataset

Synthetic bugs

A famous problem with evaluating dynamic analysis techniques is suitable datasets with ground truth variables. Using software listed in common vulnerabilities and exposures database could provide valuable test data, knowing a bug exists in a real-world software version; however, a challenge of performing large-scale analysis on multiple programs. Furthermore, there are no standards for describing a bug, making it time-consuming to analyze the software and how the bug can be triggered; moreover, triaging the results manually as there are no ground truth variables.

LAVA

LAVA [30] tries to mitigate the abovementioned issues by synthesizing bugs and injecting them into the source. Each injected bug has an identification number serving as a ground truth variable. Furthermore, we know what kind of bug it is and where it is. The concept of LAVA is to keep it simple, where each bug should have one triggering input, the bug should be as realistic as possible, and the effect of the bug should trigger behavior such that it affects the program's security.

LAVA uses static and dynamic analysis to find data an attacker can control. The dynamic taint analysis uses what they refer to as *liveness*, which refers to a particular byte in an input that decides what branches are taken; doing so captures how a particular byte affects the control flow. Modifying that value is part of generating bugs. Another metric *Taint compute number* measures the amount of computation performed on a variable at the current point in the program. Having both of these measures combined gives out something called ***D**ead **U**ncomplicated and **A**vailable data* (DUA); the DUA is at that point in the execution where specific byte(s) can be used to trigger a bug without changing the normal program control flow. When DUA is discovered, a dataflow relationship is created between the DUA and the attack point. The DUA bytes have to represent a particular value, such as not triggering the bug all the time and emulating real-world bugs. The DUA must be able to affect the program at the time of the attack point by doing operations that will lead to a bug (i.e., buffer overflow). Furthermore, the DUA can be stored in a global variable, so it will be available at the attack point if it is out of scope.

So in a high-level view, clang[4] is used to instrument the source code with taint queries. Then PANDA[5] is used to record the program with a particular input, try to discover the vulnerable parts (i.e., DUA), then use clang again to inject the bug into the program source.

Let us assume we do a taint analysis of a particular program **A**, which

[4] https://clang.llvm.org/
[5] https://panda.re/

at line 214 has a buffer where four bytes represent a DUA. Then, later on, the bytes stored in the global variable could be used to corrupt pointer *C*, causing an out-of-bounds read.

There were modifications done to LAVA bugs to increase bug realism. Sridhar [64] improved LAVA by not storing the DUA in a global variable but instead passing the DUA by reference between functions. So, after the DUA is stored and the attack point(when DUA is used to corrupt a pointer).

LAVA only supports injecting buffer overflow type of bugs [30]. The type of bug in the LAVA dataset are *memory under use(**MUS**)* type of bugs according to our memory bug model defined in 2.9.1, the *cause* being a wild pointer, the improper operating was a read and *consequence* resulting in a buffer overflow.

Weaknesses with synthetic bugs

There is some criticism of synthetic bugs, issues such as *limiting types of bugs*, *bug realism*, and *ecological validity* were raised by Bundt et al. [12]. The way LAVA injects bugs into code (i.e., DUA) limits the synthetic bug types into buffer overflows. The *realism* of synthetic bugs where injected code with auto-generated variable names differs from code with organic bugs. This might affect a program's data and control flow; fuzzers could be optimized to find such semantic [12]. Another issue is that LAVA bugs rely on global variables and magic bytes to trigger the bug [30]. In addition, the simplicity of the data flow usually differs from organic bugs; this could also taint the evaluation of automated bug-finding techniques. Also, the distribution of LAVA bugs on the ″main path"; ′main path" defined as the median discovery time of edges that are less than one hour., makes it easier to discover by a fuzzer [12].

Even though it seems that LAVA bugs are easier to find than organic bugs [12], we argue that using real-world programs and synthetic bugs would still be enough to evaluate our approach. Even with the weaknesses mentioned in the previous section, the evaluation between the dataset with synthetic bugs and real-world bugs should show a similar trend. Meaning

exposing bugs could happen faster in the LAVA dataset; however, the real-world programs have more complex bugs, different types of bugs, and are harder to find. The results should, then, be ecologically valid.

The files we will evaluate are selected challenges from the **Rode0day** [33] competition. Rode0day was a bug-finding competition that produced a set of buggy programs as challenges for competitors to compete. It was a joint effort with researchers from MIT, MIT Lincoln Laboratory, and NYU. The competition aimed to evaluate and learn about the different bug-finding techniques. They primarily used LAVA to inject the synthetic bugs; however, there were also manually injected bugs along with Angora [58] bugs.

3.4 Experimental design

We choose a comparative experimental research design to determine if the static analysis is improving the current hybrid way of automatic vulnerability detection, i.e., hybrid fuzzing. All targets will be fuzzed over a six hour period.

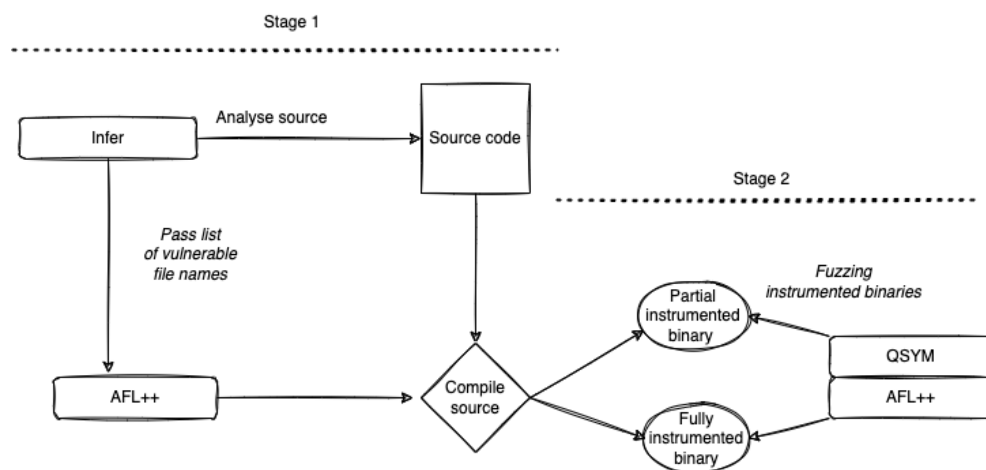

Figure 3.6: Overall view of how the tools are used together

Figure 3.6 shows the overall design of the tools used together.

The process happens in two stages:

- **Stage one**: download non-configured buggy source code, then we configure the source (i,e. capture compiler path, libraries), *infer*

performs its capture and analysis phases. Finally, the analysis results get stored in a directory, along with logs and other information about the process. File paths where *infer* marked possible buggy code locations are written to a text file.

- **Stage two**: reconfigure the source as the singularity images differ between the analysis tools. First, export the text file with file paths as an environment variable; we compile the source to partial instrument the binary(see section 3.4 for details) first. Next, unset the environment variable with the text file and compile and instrument the entire binary. Three processes are launched for the dynamic analysis: first, on the fully instrumented binary, one AFL++ fuzzing, and one hybrid fuzzing with AFL++ and QSYM. Next, hybrid fuzzing with AFL++ and QSYM on partial instrumented binary. Each process is run for six hours, and memory, time limit, and paths are given in the JSON file format.

Partial and full instrumentation

The **instrumentation** done by AFL++ is the pivotal point when we use static and dynamic analysis together in contradiction to only dynamic analysis. A quick recap, by *instrumentation*, we refer to the injection of a random value AFL++ injects at the start of a basic block. This is how AFL++ records coverage by evaluating an input's edge coverage and prioritizing the inputs that show they get an increase.

The way we do that is first to analyze with *infer*; then, after *infer's* analysis is concluded, a file with the vulnerable file names is generated. Then, AFL++ uses an environment variable with a path to the file and instruments said filenames only. This generates the partial instrumented binary; then, we instrument across all files to generate a fully instrumented binary. Since the AFL++ is an evolutionary coverage-guided fuzzer, inputs generated while fuzzing the partial instrumented binary will be drawn towards the part of the code *infer* marked as vulnerable. Creating a

fully instrumented binary requires us to instrument every basic block, and tracing the coverage-gaining inputs will have a performance cost. However, even though we lose the ability to favor inputs that cover a broader part of the entire program in producing a partial instrumented binary, we gain in efficient and targeted fuzzing. For example, imagine if we have a program **A** that has a total of 2000 basic blocks. Let us say the functions *infer* marked consist of about 600 blocks total. Now, if we had fully instrumented the binary, the randomness while fuzzing could draw the fuzzing away from the functions; we want to explore because of *infer's* analysis. By only partially instrumenting, we improve our chances of favoring the inputs we need and disregarding the inputs we do not necessarily need.

Overall, partial instrumentation is low cost, implementation-wise, and regarding resource consumption during fuzzing. This is because all we need to do is pass a list of file names when compiling the source.

Infer's issues

How *infer* [37] reports bugs depends on the options you use for the analysis. Although there are several defaults and non-defaults, we choose two non-defaults that only focus on memory safety issues; *Pulse* and *bufferoverrun*.

Pulse performs memory safety analysis. It reports issues such as memory leaks, constant address dereference, null dereference, and more. The *Pulse* option only reports when all conditions on the path are true.

Bufferoverun or InferBO performs out-of-bounds array access. InferBO calculates an array's bounds of size and offset and reports access violations. There are several levels of warnings reported, L1 through L5 and U5. L1 is likely a true bug, for example, if the array size is [1,4] and the offset is [5,5]. U5 is when the values are unknown, and S5 is when the values are symbolic [37].

Chapter 4

Results

In this chapter, we will describe our experimental results conducted on both LAVA and CVE datasets and answer our research questions. We first present the results form the LAVA dataset, CVE dataset, and finally research questions.

4.1 LAVA files

In Figures 4.1 and 4.2, we refer **partial_crash** to crash where the executable is partially instrumented based on *infer* results. **Partial_LAVA** refers to when the crash is an actual bug, meaning the crash can be confirmed with bug ID. **Full_crash** and **full_LAVA** are when the executable is fully instrumented. Furthermore, in Table 4.1 multiple inputs can crash on the same LAVA bug, creating duplicates, and discovery time records the first input exposing an LAVA bug.

Software	Number of bugs	bugs found Partial instr	bugs found Full instr	Discovery time
jpegS	125	46	30	85 sec[1]
yamlB4	5	12	10	25 sec[1]
tinyexprB2	4	0	0	N/A[2]

[1] with partial instrumentation [2] crashes couldn't be verified with bug ID

Table 4.1: LAVA experimental results

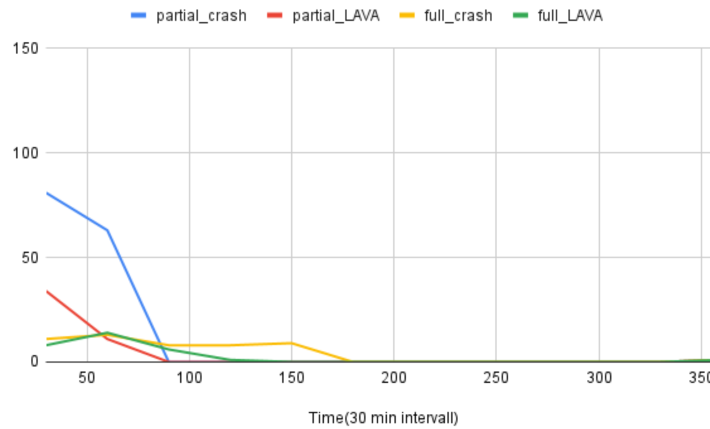

Figure 4.1: *jpegS* dynamic analysis results.

jpegS:

After running *infer* over the source code of *jpegS*, it marked 23 of 69 files as vulnerable. The analysis took 36.34 seconds across 32 cores and averaged 277.1 GB of memory. It found 102 issues with the code, mostly integer overflow issues. After generating the list of files marked as vulnerable, we instrumented the executable in those places and ran the dynamic analysis phase. In the files marked by *infer*, there are 56 LAVA bugs. Hybrid fuzzing of the partially instrumented binary produced 143 crashes(SIGNAL 11, 9, and 6), and bug ID could confirm 46 crashes. There are 125 LAVA bugs in the entire source, and fuzzing with AFL++ alone on a fully instrumented binary could not produce any crashes. However, with hybrid fuzzing, that is, AFL++ and QSYM, 52 crashes and 30 confirmed with bug ID. Furthermore, in Figure 4.1, we can see that fuzzing on a partial instrumented binary shows better performance early in the fuzzing.

yamlB4:

Compared to jpegS 69 files, yamlB4 is smaller with 9 files. *Infer* took 33.7 seconds using 64 cores and had 2TB available. 7 of 9 files were marked as vulnerable by *infer* with 100 issues. There were a total of 5 LAVA-bugs in all 9 files, and 1 LAVA-bug in the files infer marked as having issues.

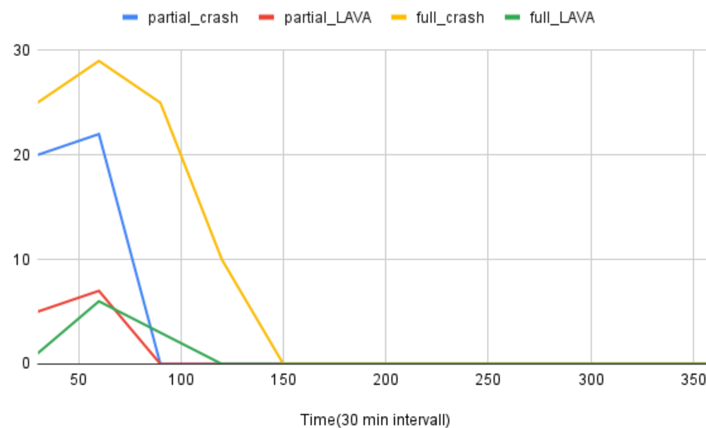

Figure 4.2: *yamlB4* dynamic analysis results.

Fuzzing the binary with AFL++, both partial and full instrumentation, did not yield any crashes. However, combining AFL++ and QSYM did produce some results. Moreover, as you can see in Figure 4.2, it did produce numerous crashes when we tested the fully instrumented binary. Using AFL++ and QSYM on the partial instrument produced fewer crashes overall; however, most were LAVA bugs.

tinyexprB2:

Infer found 11 issues with the code; the analysis phase took 5.5 seconds, averaging over 32 cores, and had 281 GB available during the analysis process. Out of the three files containing bugs, it marked two as possibly containing bugs. During the dynamic analysis, using ALF++ on the fully instrumented binary did not produce any crashes. On the other hand, combining AFL++ and QSYM did produce 43 crashes when the binary was partially instrumented and eight crashes on the fully instrumented part. However, we could not confirm that any of the crashes on partial instrumented binary and fully instrumented binary was because of a LAVA bug. However, the crashes were all reported as signal 11, meaning the program tried to access memory not belonging to the current process.

4.2 CVE files

gifsicle

Gifsicle is one of the real-world programs where fuzzed. The source files contain a null dereference, and after running *infer* over the source code, it did identify the bug; after the analysis phase had a duration of 33.19 seconds. *Infer* analyzed the code over 32 cores on average with 280 GB of memory available. After performing full instrumentation of the binary with AFL++, AFL++ found 14.9 crashes on average after a fuzzing campaign of 6 hours, and it produced a crash 12.6 minutes into fuzzing because of the CVE.

Combining AFL++ and QSYM, the number of crashes increased to 39 crashes on average. However, the first instance of a crash that was because of the CVE took 15.8 minutes compared to AFL++ alone, 12.6 minutes.

Now, fuzzing on the partially instrumented binary had a lower number of crashes than fuzzing on a fully instrumented one; however, the first crash because of CVE happened 11 minutes fuzzing campaign with AFL++ and QYSM. A full 1.6 minutes before AFL++ in a fully instrumented binary and 4.8 minutes before the hybrid version(i.e., AFL++ and QSYM).

yara

As mentioned earlier, it has a bug that can cause a stack overflow. This time we ran *infer* over the source files resulting in an analysis phase that elapsed 39 seconds, averaging over 64 cores with 2 TB of memory available. *Libyara* contains a source file called *re.c*, where there is a function _yr_re_emit which tries to emulate a regular expiration engine. The function is called _yr_re_emit recursively and infer found a memory leak that could lead to triggering the bug contained mentioned in the CVE. After 1.9 hours in a 6-hour fuzzing campaign with AFL++, it did trigger the bug. However, only on a fully instrumented binary. AFL++ and QSYM did not find the bug, regardless of whether it was on a fully or partially instrumented binary.

file

The Unix file command contains a stack overflow [21]. *Infer* analysis phase took 14.9 seconds, averaging over 32 cores. During *infer's* analysis, it did not locate the local buffer with the vulnerability. However, it did mark *mget()* in *file/src/soft magic.c* as having a buffer overflow bug. Since *file* tries to match in the database, using *mget()* can expose the vulnerability. However, we could not expose the bug after instrumenting(both partial and full) the binaries and performing the dynamic analysis.

openjpeg

Infer analysis took 52 seconds and averaged over 32 cores; during the analysis phase, *infer* marked the variable l_data_size that is the cause of the off-by-one error as vulnerable to buffer/integer overflow; if the variable was passed to allocating functions that cause the vulnerability.

Furthermore, the dynamic analysis found the bug only if the binary was fully instrumented, specifically, hybrid fuzzing(i.e., AFL++ and QSYM).

lepton

The analysis of lepton took 4 minutes, averaging over 32 cores. Again, infer marked variables such as *luma_splits* and functions related to exposing the bug. However, neither the fully instrumented nor partial instrumented binary could expose this bug.

4.2.1 Answering the Research Questions

In this section, we address the RQs described at section 1.4.

4.2.2 Research question 1

RQ1: *What are the interesting hybrid combination of fuzzing and analysis techniques?*

Both in our empirical evaluation and other works done previously [12, 73], hybrid fuzzing with concolic execution and fuzzing outperforms tra-

ditional fuzzing in most cases. However, it is more expensive concerning resource usage, seeing that constraint solving is computationally expensive. There is work that combines static and dynamic analysis [16, 39, 44], that seems to improve the overall performance.

Our empirical results seem to show that combining the two analysis techniques improves detection time but not the overall number of bugs found. Now, this is to be expected. In combining static and dynamic analysis, we favor inputs that increase code coverage in only the parts of the code where the static part is deemed vulnerable. Combining AFL++ and QSYM did show an improvement in discovery time when evaluating partial instrumented binary in the LAVA dataset and *gifsicle*. However, fuzzing a fully instrumented binary outperformed fuzzing a partially instrumented in the real-world experiment in the number of bugs found overall.

Fuzzing a partially instrumented binary does not seem to increase a specific type of bug class either. However, our datasets contain mostly *MUS* class bugs, and performing experiments on datasets with a more balanced number of bug classes could give more generalizable results.

4.2.3 Research question 2

RQ2: *Can we find the same occurrences of bugs with static code analysis tools and dynamic analysis ?*

Here we will address RQ2 and what we mentioned in section 2.9.2. The bugs in *gifsicle*, *proftpd* [27], *vim* [26], and *yara* were discovered by AFL++. QSYM found the bugs in *file*, *openjpeg*, *FFmpeg* [23], and *lepton*. We can see the distribution bug classes in table 4.2. The definitions for bug classes (i.e, **MAD**, **MAL**) can be found in section 2.9.1

Infer did find almost all bugs, both found by QSYM and AFL++, more specifically, **75%** of all bugs across all files. This could indicate that *infer* can find the same bugs. A reason could be that the heap and shape analysis performed by *infer* [15, 54] is better at finding *MUS* bugs since most bugs in the dataset are *MUS* class. Since the pre and post-conditions needed

Memory bug classes			
MAL	**MAD**	**MUS**	**MDL**
yara	openjpeg lepton	LAVA file gifsicle proftpd FFmpeg	vim

Table 4.2: Bug class distribution

to expose, mostly MUS bugs could more often than not be in the same memory footprint; however, the results here are insufficient to draw that conclusion.

An interesting observation is that most of the bugs across our dataset are of the class *memory under use (**MUS**)*, we can see that in Table 4.2. This could indicate that fuzzers and hybrid fuzzers tend to find such bugs. The reason for this could be because a **MUS** bugs operation is to *read*, *write* are easier to ′guess" when mutating. We can see this trend in other research as well; for example, the top two reasons behind CVE's that OSS-fuzz found were *dangerous memory write* and *dangerous memory read* [29].

The bugs found by QSYM are bugs where fuzzers failed, such as *openjpeg*. OSS-fuzz could not find the bug because it could not meet the complex constraints needed to expose this bug [73]. However, that is not needed when analyzing with *infer*. *Infer* marked the variable *l_data_size* that is used for the buffer and other parts in the code as possible to of a buffer overflow. This could indicate that static analysis could handle finding complex bugs where fuzzers fail. However, seeing that when dynamic analysis reports a possibility of a bug, it usually crashes. While a static analyzer could only report a possibility of a bug, and the resulting information may be invalid without any ground truth variable or previous information about an existing bug.

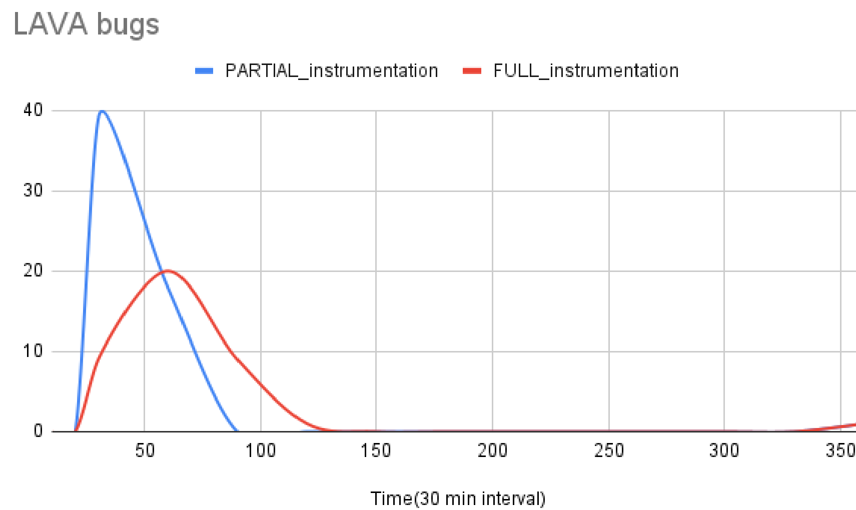

Figure 4.3: Graph representing bug discovery over time for the LAVA-dataset.

4.2.4 Research question 3

For **RQ3**: *How well do the different approaches compare in terms of efficiency and bug discovery on real world software?*

We can look through our results we present in Table 4.3. In the cases where both full and partial instrumentation's of binary exposed the bug, fuzzing using partial instrumentation found the bug faster than fuzzing with full instrumentation. We can see that in both the LAVA experiment and real-world programs. However, overall the full instrumentation exposed more bugs in most cases. For example, in Table 4.3, we can see that a partial instrumented binary outperforms the fully instrumented binary when evaluating the LAVA dataset.

An interesting observation is that when we fuzzed *openjpeg* the fully instrumented binary, we could observe the crash caused by the bug. However, no crash exposed the bug with fuzzing partial instrumented binary. This could indicate that the partial instrumented binary missed some dependencies by not instrumenting the whole binary, which leads to input generation favoring inputs that could trigger the bug. Another reason could be that the fuzzer missed the crash since we forward crash dump notifications to an external utility, AFL++ could evaluate genuine

Software	Type of bug	Partial instr	Full instr	Discovery time
jpegS	LAVA	Number of bugs: 46	Number of bugs: 30	85 sec[1]
yamlB4	LAVA	Number of bugs: 12	Number of bugs: 10	25 sec[1]
tinyexprB2	LAVA	Number of bugs: 0	Number of bugs: 0	N/A
file	CVE	×	×	N/A
gifsicle	CVE	✓	✓	11 min[1]
yara	CVE	✓	×	1.9 hours[2]
openjpeg	CVE	✓	×	4.3 min[2]
lepton	CVE	×	×	N/A

[1] with partial instrumentation [2] full instrumentation

Table 4.3: LAVA results (note that multiple inputs can crash on the same LAVA bug, creating duplicates) and CVE results

crashes as timeouts. This could be rectified by modifying the kernel core pattern as root, which we could not do.

Overall the eight files we evaluated, fuzzing the partial instrumented binary could expose bugs 37.5% of the time, while fuzzing on a fully instrumented binary had 62.5% discovery rate.

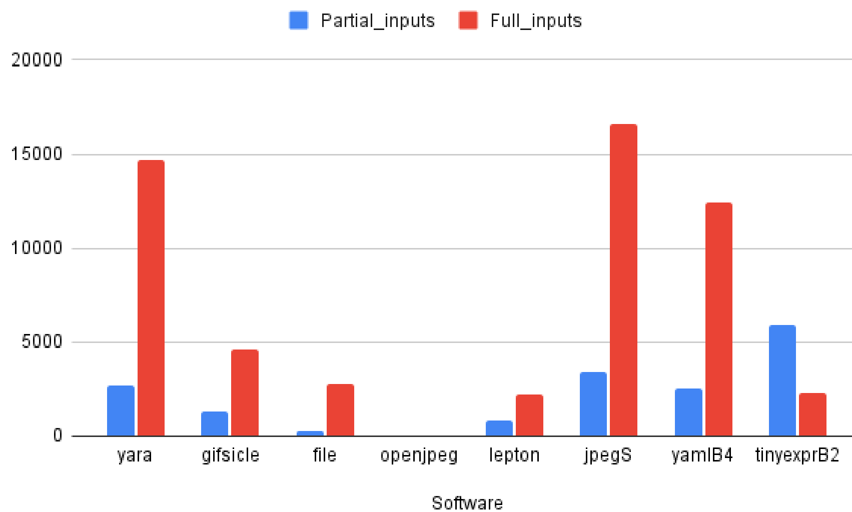

Figure 4.4: Graph depicting difference number of test cases generated under fuzzing.

Figure 4.4 shows performance difference in inputs generated, the num-

ber of generated inputs saw a substantial decrease with our approach. **Partial_inputs** refers to inputs generated by fuzzing a partially instrumented binary, and **Full_inputs** is when inputs are generated by fuzzing a fully instrumented binary. The details about partial and full instrumentation is defined in section 3.4.

The difference in instrumentation pushes the fuzzing to value inputs that increase coverage in fewer files when partially instrumented. It would stand to reason that this narrows the space in which mutated inputs could increase coverage, resulting in fewer inputs overall. We see that as well; across all experiments, when the binary is partially instrumented, the number of generated inputs saw a 67.4% decrease as opposed to full instrumentation. However, if we look at crashes, the decrease in crashes only saw a 34.1% decrease when we compare the results from fuzzing full and partial instrumented binary. Another interesting observation we can make is that in Figure 4.4, we can see that LAVA experiments produce more inputs than CVE experiments, even with partial instrumentation. The trade-off in coverage overall does not impact the ability to produce interesting inputs; this could also speak to the efficiencies difference between the two approaches. The tracing done by the fuzzer could cause unnecessary overhead, seeing that most inputs generated do not increase coverage, work done by Nagy and Hicks [52] show that less than 1 in 10 000 generated input actually increase coverage.

4.3 limitations

In all research there are limitation and possible weaknesses that could be improved upon. This section will attempt to show what the limitation are.

User privileges

Using Ex3 posed challenges, such as not having root access, and forced us to use singularity images to be able to use different environments required to use the tool(i.e., infer, AFL++ and QSYM). However, not having

root access comes with its own complications, one being increased complications when configuring each target. For example, infer's singularity image is based on Debian OS, AFL++ is based on Ubuntu 18.04 LTS, and QSYM is based on Ubuntu 16.04 LTS(because of QSYM dependencies on old pin versions [73]). However, each image used the host systems kernel, which is Linux 4.15.0-156-generic, infer, and AFL++ analysis is independent, and QSYM is designed to work with a fuzzer. Furthermore, the different source codes for the targets required different packages and compilation commands. This forced us to manually compile each of the targets to create the binaries.

Another limitation of user privileges is that since the system sends the core dump notifications, an external utility and getting this information through API can cause legitimate crashes to be misconstrued as timeouts instead. Meaning we might lose some crashes that could be valuable.

The issues mentioned above could impact our analysis results and have influenced what our targets were.

Simple design architecture

The design of combination infer with the hybrid fuzzing was by choice; the information exchange between the tools relies almost entirely on the file system. This way might be lightweight, however, infer's analysis data and how it reasoned about the vulnerabilities might be valuable to the fuzzing process. For example, while analyzing openjpeg, infer did capture the heap overflow bug. Nevertheless, fuzzing the partial instrumented binary did not produce a crash because of the bug, and fuzzing the full instrumented binary did. Some dependencies might have been needed for the bug to be triggered that were not covered by any inputs generated while fuzzing the partial instrumented binary.

Due to the simplicity of our experimental design, we focus on C/C++ code exclusively and only on x86-64 Linux distributions. Even though Java is an imperative program language, the garbage collector might significantly impact the memory bug model and the combination of static

and dynamic analysis.

Other types of bugs

We chose to focus this thesis on memory bugs in this thesis. Concurrency, algorithmic bugs, performance bugs, and all other types of bugs are valid concerns regarding automatic bug detection. The results presented in this thesis do not contribute to understanding how and if combining static and dynamic analysis will positively impact efficacy concerning finding bugs other than memory bugs.

Many of the mentioned limitations in this section boil down to the desire to keep the experimental design simple and produce a sort of "proof-of-concept" for our research objectives.

Chapter 5

Conclusion

5.1 Summary

Software defects are and always have been a challenging problem in software development. Causing security issues, billions of dollars lost, and countless hours for the few security researchers. In 2016, DARPA launched the cyber grand challenge(CGC) to learn more about the feasibility of finding and patching software defects using automated techniques. Hybrid fuzzing was a central technique used in the CGC and has been a hot topic in academic research for the last few years. However, applying hybrid fuzzing in the industry is a non-trivial task, and research into optimizing dynamic analysis is still ongoing. In this thesis, we set out to investigate if we could integrate static analysis with hybrid fuzzing for efficient bug detection. We aimed to use the scalability of static analysis to target the part of the code where the static analysis is deemed vulnerable. We empirically evaluated a dataset containing synthetic bugs [30] and a dataset with real-world bugs taken from reported CVEs. We also classified the bug in both datasets with a memory bugs model [11] to see if we could learn what type of bug each technique usually could find and if we could use that knowledge to integrate the techniques better. We learned that our approach usually could find a bug faster. However, the overall number of bugs is still greater when the dynamic analysis covers the whole program, 62.5%

to 37.5%, the former being the number of bugs found by using dynamic analysis in the traditional way and the latter only focusing on vulnerable" parts. The number of input seeds generated during fuzzing is also reduced by 67.4% when we use a partially instrumented binary, yet the crashes were only reduced by 34.1%, compared to fuzzing a fully instrumented binary. This shows us that our approach produces less uninteresting input seeds.

5.2 Suggested Future Work

We mentioned in section 4.3 that losing the information gathered during the static analysis phase could be valuable for the dynamic analysis phase. There are other works that leverage static analysis more comprehensively than presented in this thesis; work that supports the combination of techniques. For example, Ji et al. [39] uses static analysis to extract semantic information about the target program to aid in AFL's seed selection. Another use of static analysis is when Li et al. [44] uses static analysis to collect comparison information to overcome magic bytes comparisons. It shows that the dynamic analysis phase could benefit by using the data gathered by static analysis. Future work could directly use the information gathered about the code during the static analysis phase. More specifically, how a particular function might contain a vulnerability, *infer* reasons about code and attempts to build proof to prove correctness, and failed proof is valuable information not directly used in this thesis. So, for example, the failed proof that was reported as a bug by *infer* could be passed to the dynamic analysis tools to build input seeds to prove a bug's existence.

Chen et al. [16] label parts of the code as vulnerable and value seeds that guide the execution toward those parts; they refer to their solution as the **bug-driven** principle. Their solution also tries to reason the feasibility of vulnerabilities labeled and generate concrete test cases. Adopting the test case generation and seed selecting strategy of Chen et al. [16], together with our list of vulnerable parts of code based on static analysis, might produce some interesting results.

The memory bug model [11] could be expanded to include other forms of bugs so that the solution could be more generalizable. There are other classifications of software bugs, like work done by Asadollah et al. [2] that built a classification taxonomy of concurrency bugs. Going over literature and building a comprehensive taxonomy that encompasses multiple types of software bugs and builds a dataset based on this could enhance our understanding of the most efficient use concerning static and dynamic analysis techniques.

Appendix A

Analysis tools

Infer

Synopsis: static analyzer to detect bugs in Java/C/C++/Objective-C code.
Version: version v1.1.0
Homepage: https://fbinfer.com/ and https://github.com/facebook/infer/

```
infer run --pulse --bufferoverrun -- make
```

Note: the argument "`-- make`" is for situations where a Makefile is present; please refer to https://fbinfer.com/docs/infer-workflow for more compilation commands.

AFL++

Synopsis: coverage guided grey-box fuzzer, community version of the original AFL.
Version: version 3.11c
Homepage: https://aflplus.plus/ and https://github.com/AFLplusplus/AFLplusplus/
Environment variables: AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES, AFL_KILL_SIGNAL, AFL_NO_AFFINITY, AFL_LLVM_ALLOWLIST , and AFL_SKIP_CPUFREQ. All fuzzing used fixed seed 42, documentation for environment variables can be found here: https://aflplus.plus/docs/env_variables/.

QSYM

Synopsis: practical concolic execution engine designed for hybrid fuzzing.
Version: commit-hash ea4a724170835336a60ec80d957b78b37651b955
Source: https://gitlab.com/wideglide/qsym
NB: QSYM relies on an old pin version for dynamic binary translation. This means QSYM works best for Linux kernel version 3.13 and is not guaranteed to produce good results for versions 4.4 or higher.

Singularity

Synopsis: container platform designed for replication and to run applications on high-performance computing clusters.
Homepage: https://docs.sylabs.io/guides/3.5/user-guide/introduction.html
Version: 3.10.0

Bibliography

[1] (ISC)². *Cybersecurity Professionals Stand Up to a Pandemic (ISC)2 CYBERSECURITY WORKFORCE STUDY, 2020*. 2020. URL: https://www.isc2.org/-/media/ISC2/Research/2020/Workforce-Study/ISC2ResearchDrivenWhitepaperFINAL.ashx?la=en&hash=2879EE167ACBA7100C330429C7EBC623BAF4E07B (visited on 02/21/2021).

[2] Sara Abbaspour Asadollah et al. "Towards classification of concurrency bugs based on observable properties." In: *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*. IEEE. 2015, pp. 41–47.

[3] Thanassis Avgerinos et al. "Enhancing symbolic execution with veritesting." In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 1083–1094.

[4] Thanassis Avgerinos et al. "The Mayhem Cyber Reasoning System." In: *IEEE Security Privacy* 16.2 (2018), pp. 52–60. DOI: 10.1109/MSP.2018.1870873.

[5] Roberto Baldoni et al. "A survey of symbolic execution techniques." In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[6] Alexandru G Bardas et al. "Static code analysis." In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107.

[7] Frank S de Boer and Marcello Bonsangue. "Symbolic execution formally explained." In: *Formal Aspects of Computing* (2021), pp. 1–20.

[8] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. “Fuzzing: Challenges and Reflections.” In: *IEEE Software* (2020).

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based greybox fuzzing as markov chain.” In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[10] Irena Bojanova. URL: https://samate.nist.gov/bf/Home/Approach.html.

[11] Irena Bojanova and Carlos Eduardo Galhardo. “Classifying Memory Bugs Using Bugs Framework Approach.” In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2021, pp. 1157–1164.

[12] Joshua Bundt et al. “Evaluating synthetic bugs.” In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 716–730.

[13] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later.” In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[14] Cristiano Calcagno and Dino Distefano. “Infer: An automatic program verifier for memory safety of C programs.” In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 459–465.

[15] Cristiano Calcagno et al. “Compositional shape analysis by means of bi-abduction.” In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009, pp. 289–300.

[16] Yaohui Chen et al. “Savior: Towards bug-driven hybrid testing.” In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1580–1596.

[17] Dulma Churchill et al. *Infer: Static Analysis at Scale (ICST2018)*. 2018. URL: https://youtu.be/3SNPmCgl5eU (visited on 06/24/2022).

[18] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. “Memfuzz: Using memory accesses to guide fuzzing.” In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 48–58.

[19] Jake Corina et al. “Difuze: Interface aware fuzzing for kernel drivers.” In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2123–2138.

[20] Domenico Cotroneo et al. “How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation.” In: *Journal of Systems and Software* 113 (2016), pp. 27–43.

[21] *CVE-2017-1000249*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000249.

[22] *CVE-2017-12762*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12762.

[23] *CVE-2017-17081*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17081.

[24] *CVE-2017-8891*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8891.

[25] *CVE-2017-9438*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9438.

[26] *CVE-2019-20079*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20079.

[27] *CVE-2020-9272*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9272.

[28] Cybersecurity and Infrastructure Security Agency. *Alert (AA20-099A) COVID-19 Exploited by Malicious Cyber Actors*. 2020. URL: https://us-cert.cisa.gov/ncas/alerts/aa20-099a (visited on 02/25/2021).

[29] Zhen Yu Ding and Claire Le Goues. “An Empirical Study of OSS-Fuzz Bugs.” In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 131–142.

[30] Brendan Dolan-Gavitt et al. "Lava: Large-scale automated vulnerability addition." In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 110–121.

[31] *EPYC 7601 - AMD - WikiChip*. URL: https://web.archive.org/web/20220615134933/https://en.wikichip.org/wiki/amd/epyc/7601.

[32] *EX3*. URL: https://www.ex3.simula.no/.

[33] Andrew Fasano et al. "The rode0day to less-buggy programs." In: *IEEE Security & Privacy* 17.6 (2019), pp. 84–88.

[34] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research." In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[35] Shuitao Gan et al. "Collafl: Path sensitive fuzzing." In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 679–696.

[36] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing." In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[37] *Infer static analyzer: Infer: Infer*. URL: https://fbinfer.com/docs/getting-started.

[38] Laf Intel. *Circumventing fuzzing roadblocks with compiler transformations*. 2016.

[39] Tiantian Ji et al. "AFLPro: Direction sensitive fuzzing." In: *Journal of Information Security and Applications* 54 (2020), p. 102497.

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

[41] Kohler. *Null-Dereference-in-apply*$_i$*mage* $- ISSUE - 130 - Kohler - Gifsicle$. URL: https://github.com/kohler/gifsicle/issues/130.

[42] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken. "Discovering software vulnerabilities using data-flow analysis and machine learning." In: *Proceedings of the 13th international conference on availability, reliability and security*. 2018, pp. 1–10.

[43] Harjinder Singh Lallie et al. "Cyber security in the age of covid-19: A timeline and analysis of cyber-crime and cyber-attacks during the pandemic." In: *Computers & Security* (2021), p. 102248.

[44] Yuekang Li et al. "Steelix: program-state based binary fuzzing." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 627–637.

[45] Zhenmin Li et al. "Have things changed now? An empirical study of bug characteristics in modern open source software." In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. 2006, pp. 25–33.

[46] Daniel Lohrmann and Dan Lohrmann. *2020: The Year the COVID-19 Crisis Brought a Cyber Pandemic*. 2020. URL: https://web.archive.org/web/20210615094118/https://www.govtech.com/blogs/lohrmann-on-cybersecurity/2020-the-year-the-covid-19-crisis-brought-a-cyber-pandemic.html (visited on 06/15/2021).

[47] Rupak Majumdar and Koushik Sen. "Hybrid concolic testing." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 416–426.

[48] Barton P Miller et al. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1995.

[49] Charlie Miller and Zachary NJ Peterson. "Analysis of mutation and generation-based fuzzing." In: *Independent Security Evaluators, Tech. Rep* 4 (2007).

[50] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[51] David J Musliner et al. "Fuzzbomb: Autonomous cyber vulnerability detection and repair." In: *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV 2015)*. 2015.

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

[53] Anh Nguyen-Tuong et al. "Xandra: An autonomous cyber battle system for the Cyber Grand Challenge." In: *IEEE Security & Privacy* 16.2 (2018), pp. 42–51.

[54] Peter O'Hearn, John Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures." In: *International Workshop on Computer Science Logic*. Springer. 2001, pp. 1–19.

[55] Sean Peisert et al. "Perspectives on the SolarWinds Incident." In: *IEEE Security & Privacy* 19.02 (2021), pp. 7–13.

[56] Van-Thuan Pham et al. "Smart greybox fuzzing." In: *IEEE Transactions on Software Engineering* 47.9 (2019), pp. 1980–1997.

[57] John C Reynolds. "Separation logic: A logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[58] Subhajit Roy et al. "Bug synthesis: Challenging bug-finding tools with deep faults." In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 224–234.

[59] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A concolic unit testing engine for C." In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[60] Team Shellphish. "Cyber grand shellphish." In: *Phrack Papers* (2017).

[61] Yan Shoshitaishvili et al. "Mechanical phish: Resilient autonomous hacking." In: *IEEE Security & Privacy* 16.2 (2018), pp. 12–22.

[62] *singularity container 2.5.1 documentation*. URL: https://web.archive.org/web/20220615140641/https://singularity-userdoc.readthedocs.io/en/latest/.

[63] Jia Song and Jim Alves-Foss. "The darpa cyber grand challenge: A competitor's perspective." In: *IEEE Security & Privacy* 13.6 (2015), pp. 72–76.

[64] Rahul Sridhar. "Adding diversity and realism to LAVA, a vulnerability addition system." PhD thesis. Massachusetts Institute of Technology, 2018.

[65] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[66] Deepak K Tosh et al. "Risk management using cyber-threat information sharing and cyber-insurance." In: *International conference on game theory for networks*. Springer. 2017, pp. 154–164.

[67] Junjie Wang et al. "Superion: Grammar-aware greybox fuzzing." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 724–735.

[68] Wikipedia contributors. *Harvard Mark II — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-January-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Harvard_Mark_II&oldid=1061116663.

[69] Wikipedia contributors. *Mars Climate Orbiter — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-January-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Mars_Climate_Orbiter&oldid=1064452183.

[70] Wikipedia contributors. *Software bug — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-January-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Software_bug&oldid=1063644799.

[71] Maverick Woo et al. "Scheduling black-box mutational fuzzing." In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 511–522.

[72] J Wright, Chris Eagle, and Tim Vidas. "Operating system doppelgangers: The creation of indistinguishable and deterministic environments." In: *BSD Magazine* 12.12 (2018).

[73] Insu Yun et al. "QSYM: A practical concolic execution engine tailored for hybrid fuzzing." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 745–761.

[74] Michal Zalewski. *american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.