The Pennsylvania State University
The Graduate School
College of Engineering

EXTENDING VULNERABILITY DISCOVERY WITH FUZZING AND SYMBOLIC EXECUTION TO REALISTIC APPLICATIONS

A Thesis in
Computer Science and Engineering
by
Eric Kilmer

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2017

The thesis of Eric Kilmer was reviewed and approved* by the following:

Patrick McDaniel
Distinguished Professor of Computer Science and Engineering
Thesis Advisor

Trent Jaeger
Professor of Computer Science and Engineering

Chitaranjan Das
Distinguished Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file at The Graduate School.

Abstract

In 2016, DARPA held the Cyber Grand Challenge (CGC) using a special execution and evaluation environment to compare the results of different techniques in automated vulnerability discovery. However, this special execution environment simplifies many of the complexities seen in real binaries on a desktop Linux system. In this paper, we augment the top-scoring, open source, vulnerability discovery component from the CGC by providing additional functionality with respect to files, file systems, and library function summaries to more effectively operate on realistic Linux binaries. We begin by transforming the CGC challenge binaries to resemble more realistic Linux binaries by way of dynamically linked standard C library functions and compiling for a 64-bit system. We then look at examples of popular Linux applications to evaluate our solution. We find that support for files is important and the lack of function summaries for C library functions and system calls limits the effective use of symbolic execution in a real Linux environment as compared with the CGC.

Contents

List of Figures

List of Tables

List of Code Listings

Listings

Acknowledgments

I would like to thank everyone I have had the chance to meet and spend time with throughout my academic career. I have learned and experienced so much.

I would like to specifically thank Dr. Patrick McDaniel for his encouragement, unending support, and enthusiasm. I would also like to thank you for introducing me to everyone working in the SIIS lab. Thank you Raquel for the encouragement, patience, and help throughout this thesis research. Without all of you, there is no way I would be where I am today – Thank you.

Thank you to my family for raising me to be the best I can be. This work would not have been possible without your guidance and support. I love you.

A special thanks to Team Shellphish and everyone who contributed to the Driller and `angr` projects. This thesis would not exist if Driller were closed source.

Dedication

This thesis is dedicated to my family for all of their time, support, and love.

Chapter 1
Introduction

1.1 Background

Binary and vulnerability analysis are popular areas of research, with many related conferences and papers published each year. With an ever-increasing number of software applications being developed and improved, humans alone are unable to thoroughly test their programs and catch bugs before being released to production. The importance of automated systems that can exercise not only portions, but the entirety of a program, is as high today as it ever was in the past. Not only does the continued development and research of these systems catch critical and dangerous vulnerabilities, they also help with program stability.

There are many different methods and techniques to discover bugs and vulnerabilities in software: fuzzing, symbolic execution, taint analysis, etc. However, there has been no standard benchmark to measure and compare the effectiveness of these research efforts. In 2016, DARPA held the first CGC to encourage the research and development of new methods and techniques for automated program, vulnerability, and exploit analyses. As a result, the CGC provided a benchmark for researchers to compare and measure the effectiveness of their research techniques and implementations.

1.1.1 DARPA Cyber Grand Challenge

With DARPA's first CGC held in 2016, increased importance has been placed on current and future research of automated vulnerability detection and analysis of computer software. Contestants were awarded millions in prize money based on

the performance of their systems. The CGC is similar to Capture the Flag (CTF) hacking tournaments and competitions where participants try to solve security and reverse engineering-oriented challenges and objectives. For instance, a team may be required to find an exploit that will extract a secret piece of information, the "flag," from a computer program which they can use to score points or make progress in the competition. In the CGC, however, instead of human participants, machines running automated vulnerability analyses and scheduling algorithms must compete against each other to score the largest number of points without any human interaction. Seven teams competed in the final round.

There are three phases, or areas, where a team earns points: (i) **Security**, through patching or detecting vulnerabilities; (ii) **Availability**, by measuring the impact a patch has on a program's performance and functionality; and (iii) **Evaluation**, by discovering and proving the existence of a vulnerability in an opponent's software.[1] These phases are conducted on "challenge binaries" (CBs): novel, compiled programs never seen before by the competitors. Each CB has one or more known vulnerabilities and flags and runs in a special execution environment, otherwise known as DECREE: DARPA Experimental Cyber Research Evaluation Environment. DECREE is an execution environment built specifically for the CGC and was designed to allow researchers to focus on their analysis techniques, instead of on the complexities of a real operating system. As such, DECREE limits CB functionality in a few important ways. DECREE is limited to 7 basic system calls, has no persistent filesystem or interaction with files, and executes 32-bit x86 instructions [2]. While DECREE was an enormous engineering accomplishment and made the entire competition possible, the differences between it and a standard Linux execution environment cannot be ignored.

Of the three scoring phases, we focus on **Evaluation**: discovering and proving the existence of a vulnerability in an opponent's software. Shellphish, a team originating from University of Santa Barbara (UCSB), found the most vulnerabilities of all 7 finalist teams by using Mechanical Phish, their Cyber Reasoning System (CRS), and Driller, their vulnerability discovery component.

[1] `http://archive.darpa.mil/cybergrandchallenge/assets/pdf/cgc-brochure.pdf`

1.1.2 Mechanical Phish and Driller

Of the 7 teams that competed in the main CGC event, the crash discovery component of the Mechanical Phish CRS, Driller, discovered the most vulnerabilities [2]. Driller was developed at the University of Santa Barbara (UCSB) by many of the same members of Team Shellphish. It is the combination of two vulnerability analysis techniques that complement each other strengths and weaknesses: fuzzing and selective symbolic execution. Figure 1.1 shows a simplified illustration of Driller. The fuzzing component is provided by American Fuzzy Lop (AFL), a popular fuzzer that has shown great success in finding real-world bugs in popular software like Firefox and Adobe Flash.[2] The selective symbolic execution component consists of `angr` [3], a binary analysis framework, developed at UCSB, with a QEMU[3] tracer that allows for symbolic tracing of the inputs generated by the fuzzer. During the symbolic tracing, Driller collects the symbolic constraints, looks for new paths not covered by the fuzzer, and uses a constraint solver, like Z3 [4], to generate new inputs that satisfy the new paths. The original Driller paper [1] provides additional details about Driller's techniques and implementation. We provide a more detailed description and background on fuzzing and selective symbolic execution in Section 2.2.

Built and developed for DECREE binaries, Driller has not been widely tested on applications that were built and developed for execution on a native Linux operating system such as Ubuntu. While `angr` and its symbolic tracing capabilities do support a native Linux operating system environment, there are currently many undocumented and unsupported mechanics which prevent easy use of Driller on whole-application analysis within a Linux environment. The CGC results prove that Driller is capable of discovering many bugs in non-trivial programs like those tested in the CGC. Therefore, we seek to extend Driller's functionality from the simplified DECREE system to a real desktop Linux execution environment with the aim of finding new vulnerabilities in popular software.

Furthermore, the Shellphish team[5] decided to release their entire Mechanical Phish CRS implementation, including Driller, as open source to be improved

[2] `http://lcamtuf.coredump.cx/afl/#bugs`

[3] `http://www.qemu.org/`

[4] `https://github.com/Z3Prover/z3`

[5] To our knowledge and as of this writing, Shellphish is the only team to fully open source their engine and solution components.

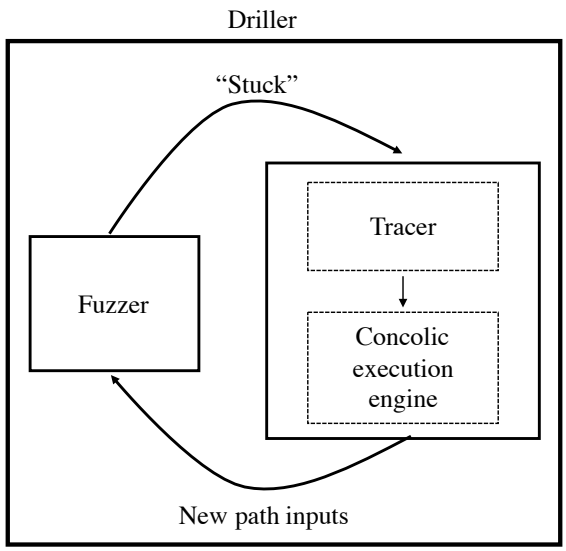

Figure 1.1: A simplified illustration of Driller's execution flow and implementation.

upon by current and future researchers and collaborators. We are in favor of this sentiment and have contributed our modifications, documentation, and datasets back to the `angr` open source repositories.[6]

1.2 Previous Work and Motivation

Driller was developed mainly with the CGC in mind, so it requires additional modifications to run effectively on any arbitrary Linux binaries. While the concept and technique for combining symbolic execution and fuzzing on large-scale binaries is not new [1, 4–6], Driller found the most vulnerabilities at the CGC and is one of the few implementations that is freely available and in active development. Fuzzing, particularly AFL and Google's OSS-Fuzz program,[7,8] has found and published a long list of discovered vulnerabilities in many different types of real open source software.[9] Fuzzers scale easily to large paths in programs with many different libraries.[10] On the other hand, symbolic execution still has many limitations when

[6] `https://github.com/angr/`

[7] `https://www.helpnetsecurity.com/2017/05/09/fuzzing-open-source/`

[8] `https://github.com/google/oss-fuzz`

[9] `http://lcamtuf.coredump.cx/afl/#bugs`

[10] Cryptographic routines and other complex checks that depend on fuzzed input hinders the effectiveness of fuzzing [7].

scaling to the large execution paths within entire programs [8]. Path explosion, floating point computation, and large constraint expressions with many variable interdependencies (like hashing and strong cryptography algorithms) are the primary limitations of symbolic execution. More details are described in Section 2.2.2 on the strengths and limitations of symbolic execution.

The work presented in this paper differs from existing full-scare binary solutions in that we extend the current state-of-the-art tool, Driller, and `angr` to support a file system with file handling, which the CGC DECREE system did not use, and implement additional C library function summaries to limit path explosion. Other open-source symbolic execution engines like KLEE [9] and S2E [10] do support files and a file system and provide system call summaries, however `angr` has other advantages over both KLEE and S2E, as described below.

KLEE only operates on an intermediate representation (IR) generated by the LLVM compiler, which requires source code. S2E is built upon KLEE with additional features to allow operation on compiled binaries, path analyzers which can be used to look for user-specified program properties, and support for both kernel-mode and user-mode Linux binaries.[11] While S2E is more mature, in development since 2011, than `angr`, in development since late 2014, `angr` is a python framework that focuses on component modularity. `angr` is able to take advantage of different sub-component implementations and output formats and interface this other components, in a plug-in fashion. Currently, `angr` supports "the 32-bit and 64-bit variants of x86, ARM, MIPS, and PPC, and offer a range of static analyses along with a powerful symbolic execution engine. This has allowed `angr` to be used as the base for an automated ROP generator, a binary patching engine, a next-generation fuzzer, an auto-exploitation engine, and other exciting stuff," with big plans for the future.[12] Acknowledging that S2E could be used as the selective symbolic execution component in a tool similar to Driller, we think `angr` is the best choice to make improvements based on `angr`'s current development rate, feature set, community involvement, choice of programming language, and the simple fact that their existing implementation of Driller is a good base to start from and has proved itself in the CGC.

[11] `https://github.com/dslab-epfl/s2e`

[12] `http://angr.io/blog/2017_01_10.html`

1.3 Thesis Outcomes

Results from the CGC showed that a vulnerability discovery technique that combines fuzzing and selective symbolic execution found the most vulnerabilities. However, the tool that implemented this promising technique, Driller, has only been tested in the minimal execution environment found at the CGC. Executing Driller with binaries in a real Linux environment requires solving several key challenges including an accurate and robust model for files and file system, and support for symbolic execution of system calls and standard C library functions. In this paper, we explore the efficacy and challenges presented in applying this joint technique to real-world Linux binaries by implementing and supporting a simple, yet effective, file system that allows Driller to support programs that utilize file-related system calls and C library functions. The effectiveness and correctness of our implementation is backed by tests provided in the widely-used GNU C library implementation. Broadly, we look at and address the following challenges that occur in realistic binaries:

- How do C library functions affect the symbolic execution stage?
- How do we best support files and a file system?
- How do we extend this technique/solution to arbitrary programs?

We evaluate our contributions first on a subset of CGC binaries that we modified to be more realistic in order to replicate the original Driller paper's results. We analyze and find a vulnerability in a real native Linux binary which uses a custom-written file parser. Furthermore, we measure code coverage and the number of vulnerabilities found, and compare these metrics with that of a freely available fuzzer, AFL, and symbolic execution engine, KLEE, for the programs in our dataset.

Chapter 2
Technical Background

2.1 Types of Vulnerabilities

There are many different types of vulnerabilities, weaknesses, and exploits that can be expressed in a computer program. There have been a few attempts to categorize the types of vulnerabilities and/or their severity: Common Weakness Enumeration (CWE) by MITRE [11], the Bugs Framework (BF) by NIST [12], and Common Vulnerabilities and Exposures (CVE) by MITRE [13], to name a few popular efforts. Referencing MITRE's CWE list of vulnerability types, a vulnerability can be classified into more than 1,000 different categories.[1]

These vulnerabilities include NULL pointer dereferences (CWE #476), cross-site request forgery (CWE #352), use of insufficiently random values (CWE #330), and execution with unnecessary privileges (CWE #250). However, our goal is not to catch all of these vulnerability types; we only attempt to discover vulnerabilities that cause a program to crash.

2.2 Vulnerability Analysis Techniques

Software developers utilize many different techniques, tools, and frameworks to prevent bugs from appearing in their code and being released to end users. For instance, many modern compilers like `gcc`[2] and `clang`[3] provide warnings about incompatible data types, uninitialized and unused variables, implicit data type

[1] `https://cwe.mitre.org/data/index.html`
[2] `https://gcc.gnu.org/`
[3] `http://llvm.org/`

casting, etc. Furthermore, if the programmer is using a modern Integrated Developer Environment (IDE), like Microsoft's Visual Studio,[4] then the IDE can automatically compile code to notify the programmer of potentially dangerous code constructs in real time.

In general, there are two types of code/vulnerability analysis: static and dynamic analysis. Static analysis techniques are performed on an unchanging, *static*, non-executable, instance of a computer program. This static instance of the program could either be source code, when available, or compiled assembly code. Both source code analysis and assembly code analysis tools have their own challenges. Source code analysis tools must process all high-level source language constructs such as classes, lambda functions, templates, etc [14, 15]. On the other hand, compiled assembly code has no high-level or abstract constructs, which makes the assembly representation more complex and difficult to follow without actually executing it.

In contrast, dynamic analysis tools operate on a changing, *dynamic*, executable instance of a program. That is, dynamic analysis tools run either part of or the entire program to examine specific properties and functionality of that program. A tool like Valgrind[5] is an example of a dynamic analysis tool. Valgrind runs the program with program inputs provided by the user and detects memory errors by tracking the uses of `malloc` and `free` memory functions.

Static and dynamic analyses have their own strengths and weaknesses. A strength of static analysis is that the techniques have access to all areas of executable code (given that source code is available) while dynamic analysis must be provided a set inputs to allow execution of all code areas. Another important difference is that even if static analysis finds a potential vulnerability, there is no guarantee that it can be exploited, while dynamic analysis (as long as it starts from the beginning of the program) can generate an input or environment configuration that can exploit the vulnerability.

While previous research combines static and dynamic analysis techniques [16–18], this paper will focus on combining two dynamic analysis techniques: fuzzing and selective symbolic execution.

[4] `https://www.visualstudio.com/`
[5] `http://valgrind.org/`

2.2.1 Fuzzing

Fuzzing is a type of dynamic execution analysis that can be considered brute-force guessing of inputs until a specified condition is met, like a program crash. There are many different ways to guess inputs, and not all of them have to be completely random. In fact, most fuzzing campaigns start with known valid inputs to the program, called *seeds*. Then, these seed inputs are mutated and modified with the goal of exercising all program functionality in ways that the original developer did not intent in order to cause a crash. For instance, a parser implementation for a specific protocol may check that a specific keyword is present and then check for a size value associated with the keyword to determine how many bytes to read. However, if the parsing implementation does not set a sane maximum value for the size, then there is a possibility that an attacker could exploit this by setting a very large size value and thus crash the program.

Popular fuzzers tend to use many different types of heuristics to avoid a worst-case brute-force time duration [19, 20]. These heuristics can be through simplistic mutations, where the fuzzer manipulates data through bit-flips, appends, insertions, and other transformations, or through a more complex generational heuristic, where the objective is to generate input data that follows the program's expected input format.

There are two main scenarios where analysts can apply fuzzers: whitebox and blackbox. During whitebox fuzzing, the analyst has access to the source code and can *instrument* the resulting compiled program to help with the fuzzing attempt. AFL is a popular whitebox fuzzer that includes its own `gcc` and `clang` compiler to insert record-keeping and logging mechanisms in order to better gauge when or how to modify a program input. On the other hand, blackbox fuzzers specialize in fuzzing programs where nothing but the compiled software is given. The techniques used by these fuzzers can combine other static or dynamic analysis methods to improve performance and results [21].

2.2.1.1 Existing Tools

There are a number of tools commonly used by analysts to conduct fuzzing campaigns or integrate into other frameworks. Three of the most popular open-source

tools are AFL, Radamsa,[6] and Boofuzz.[7] A commercial fuzzing framework, Peach Fuzzer, is also available.[8] Each of these fuzzers touts a number of features or design decisions that make them viable for different fuzzing campaigns. AFL and Radamsa even include a list of bugs found using their implementations.[9]

All of these tools are designed in a way to easily allow the fuzzing of many different types of programs. AFL's goal is to get the user up-and-fuzzing in an efficient way with as little pre-configuration or special setup as possible. Radamsa was developed primarily as a black-box fuzzer to explore file formats and protocols using various heuristics and input mutations. Boofuzz focuses on easy extensibility for customized fuzzing. Peach Fuzzer uses *peach pits* that contain test definitions and specifications for protocols commonly used in software programs.

There is more to be said about the heuristics and techniques used by each tool to speed up progress, however this is outside the scope of this paper. Please refer to each tool's documentation to learn more.

2.2.1.2 Limitations

The most critical factor in achieving the best fuzzing results lies in the number and variability of *test cases*. A test case is often an input into the program that exercises a program's functionality. It is important to have many different test cases that each test a different part of the program because this will allow the fuzzer to explore and validate more of the program's features through its input-modification techniques.

For example, the JPEG image format[10] specifies that every JPEG file starts with the magic hex bytes `ff d8 ff`, which are then followed by specific bytes at context-dependent offsets. These specific bytes act as markers or flags to tell the JPEG parser how to interpret the image data. For instance, `ff c0` is the beginning marker to indicate that information about the width, height, number of components, and component subsampling follows after. Without providing a few valid JPEG

[6] `https://github.com/aoh/radamsa`

[7] `https://github.com/jtpereyda/boofuzz`

[8] Peach Fuzzer also has a "Community Edition" which updates at a slower pace than the commercial version and is also missing some features. `https://www.peachfuzzer.com/resources/peachcommunity/`

[9] Bugs discovered with AFL, `http://lcamtuf.coredump.cx/afl/#bugs`, and Radamsa, `https://github.com/aoh/radamsa#some-known-results`.

[10] `https://www.w3.org/Graphics/JPEG/itu-t81.pdf`

files, the fuzzer could take a substantial amount of time to figure out the required bytes values to make it past the JPEG specification's magic bytes, which is one of the first checks done before processing the rest of the file.

2.2.1.3 Fuzzing Strengths

Fuzzing's strengths come from the relatively fast speed by which new inputs are tested and mutated/generated. The fuzzed program runs new inputs directly, and the fuzzer performs lightweight heuristic modifications on inputs that have produced new paths in the past. There are many different techniques and methods designed to find bugs in a more efficient manner compared to purely random methods [21].

For instance, AFL uses compile-time or virtualized (QEMU) instrumentation to track code coverage when generating new inputs and determine which inputs are interesting.[11] AFL also has the option to include strings found within the program's binary to guess new magic keywords. Peach Fuzzer uses Peach Pits[12] to guide the fuzzer in generating specification-compliant inputs that can test implementations according to the published protocol specifications.

2.2.2 Dynamic Symbolic and Concolic Execution

Symbolic execution is the process of executing a program where unknown inputs are replaced by symbolic variables. As a symbolic execution follows an instruction path, symbolic expressions are created by adding equivalent constraints and operations performed by the instructions. These variable expressions and constraints can be used to determine the concrete values of an input that will execute along the chosen code path. Symbolic execution becomes *dynamic* (DSE) or *concolic* when, instead of all inputs being replaced by symbolic variables, only specific parts of an input or specific input sources are symbolized and other values are taken from a concrete execution. Concolic comes from the combination of *conc*rete and symb*olic*. In this paper, when we refer to selective, dynamic, concolic, or plain symbolic execution, we are referring to concolic execution that uses concrete values for all values other than specified symbolic data sources. Symbolic execution is more easily understood through an example.

[11] `http://lcamtuf.coredump.cx/afl/technical_details.txt`
[12] `http://www.peachfuzzer.com/products/peach-fuzzer/peach-pits/`

Consider code Listing 2.1. The objective is to find 4 numbers that satisfy the conditional statement on line 12 and have the program crash on line 14. It is not easily apparent that one potential solution is to let $a = 40, b = 68, c = 352, d = 270$. To figure out this example by hand would be tedious and time consuming. Furthermore, fuzzing would be forced to randomly guess the whole input space of positive integers for each of the 4 inputs. However, we can use symbolic execution to solve this quickly. If we look at the corresponding constraint Table 2.1, we can use an automated satisfiable modulo theory (SMT) solving tool like Microsoft's Z3[13] to obtain a(n) solution(s). SMT solvers like Z3 are the backbone of symbolic execution tools and allow us to generate concrete values for unknown variables by solving the symbolic expressions obtained from the instructions in a program trace.

Listing 2.1 : DSE Example Source Code.

```
1  int main() {
2      int a, b, c, d;
3      printf("Please enter 4 positive
4          non-zero numbers: ");
5      scanf("%d %d %d %d", &a, &b, &c, &d);
6
7      if (a > 0 && b > 0 && c > 0 && d > 0) {
8          int x = a / 5 + b / 2 + d / 6;
9          int y = b / 4 + c / 4;
10         int z = a / 2 + d / 3;
11         if (x == 'W' && y == 'i' && z == 'n')
12             printf("You win!\n");
13             <program_crash>
14         else
15             printf("Guess again.\n");
16     }
17     else {
18         printf("Must enter 4 positive
19             non-zero numbers.\n");
20     }
21 }
```

[13] https://github.com/Z3Prover/z3

Variable	Constraint	Line Number
a	$a > 0$	8
b	$b > 0$	8
c	$c > 0$	8
d	$d > 0$	8
x	$x == a/5 + b/2 + d/6$ $x == 87$ ($x ==$ 'W')	9 12
y	$y == b/4 + c/4$ $y == 105$ ($y ==$ 'i')	10 12
z	$z == a/2 + d/3$ $z == 110$ ($z ==$ 'n')	11 12

Table 2.1: Constraints needed to reach line 14 in Listing 2.1

2.2.2.1 Existing Tools

There are many tools that implement symbolic and concolic execution. A few notable projects are Triton,[14] `angr` [3], KLEE [9], S^2E [10], SAGE [4], and Mayhem [22]. Each of these was developed with different use-cases and design decisions in mind.

All of the mentioned tools except SAGE started as academic projects. KLEE was first published in 2008 from Stanford. KLEE is "capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs." KLEE requires the presence of source code to generate these tests and achieve high code coverage. On the other hand, S^2E (Selective Symbolic Execution) was first published in June 2009 and built upon KLEE. S^2E was used to "develop a comprehensive performance profiler, a reverse engineering tool for proprietary software, and a bug finding tool for both kernel-mode and user-mode binaries." However, S2E specializes in its "ability to scale to large real systems, such as a full Windows stack."

Around the same time as KLEE, in 2008, Microsoft published details of its SAGE project which combines symbolic execution and fuzzing to discover bugs in its Windows products. SAGE is able to process programs both with and without access to source code. While SAGE is proprietary, Microsoft has announced Project

[14] `https://triton.quarkslab.com/`

Springfield[15] where they will offer SAGE and other program analysis tools as a paid, commercial service. Similarly, in 2012, researchers from Carnegie Mellon University published a paper on Mayhem, a closed-source symbolic execution tool. The team behind Mayhem has since started ForAllSecure where they plan to offer consulting services in vulnerability detection using Mayhem.[16]

Finally, `angr` and Triton were released as binary analysis frameworks with symbolic execution engines by UCSB in late 2014 and by Bordeaux University and Quarkslab[17] in 2015, respectively. These projects encompass more than just symbolic execution. Triton is able to perform taint analysis, an analysis technique described here [23], and hook into different tracers like QEMU, Intel PIN,[18] and DynamoRio.[19] The `angr` project also includes some static analysis tools to generate control flow graphs, perform value-set analysis to create value flow graphs for each function, data dependency graphs, backwards slicing, and more.[20] Additionally, `angr` is written purely in Python, which makes it easier to hack and extend but a little slower than implementations written in C/C++ like KLEE.

2.2.2.2 Limitations

While symbolic execution and constraint solving is very promising for finding bugs guarded by complex constraints and exploring a program, there are limitations involved during full scale binary analysis. One of biggest concerns is path explosion. Path explosion occurs during the symbolic execution of large or complex programs when trying to symbolize all possible paths that inputs can take. In the case of unbounded loops, an infinite number of different paths may exist. A large number of paths can also lead to complex symbolic expressions with a large number of constraints. SMT solvers like Z3 are always improving in performance through heuristics and other means. More detailed and technical information about the operation of SMT solvers can be found here [24].

One real-world example where symbolic execution will fail is when trying to

[15] `https://blogs.microsoft.com/next/2016/09/26/microsoft-previews-project-springfield-cloud-based-bug-detector/`

[16] `https://forallsecure.com/`

[17] `https://triton.quarkslab.com/`

[18] `https://software.intel.com/en-us/articles/pintool`

[19] `http://www.dynamorio.org/`

[20] `https://docs.angr.io/docs/analyses.html`

recover the input used for an arbitrary CRC32 checksum.[21] The reason for this is because the CRC32 algorithm performs many mathematical and logical operations using all input bytes to affect all output bytes. This is also true for many other secure cryptography and hashing algorithms. Thus, if we want to symbolically execute a program, it is necessary to skip these algorithms through a patch or other means.

2.2.2.3 Strengths

Although the complexity and size of a program can limit the practicality of DSE, the potential gain from DSE is worth the effort. In cases where a human is able to apply symbolic execution to small areas of a program, the result can be the simplification of obfuscated code[22] or to reverse engineer the Petya ransomware's custom cryptography.[23]

Symbolic execution provides an accurate and provable method of generating new inputs that will satisfy the constraints placed on a specific execution trace. In the case of Driller, DSE is used to find new paths in the program by inverting the original traced constraint. Discovery of new paths allows us to test more portions of the program to find deeper hidden bugs.

2.3 Vulnerability Analysis of Production Software

For a vulnerability analysis tool or technique to be useful, it must be effective for real software in a real operating system or environment. The CGC allowed researchers to develop a solution that could detect, exploit, and patch security vulnerabilities in a fully automated way without worrying about the complexities and nuances of real operating systems and environments. The CGC presented contestants with many challenges and it heightened the importance of fully automated analysis solutions. Some teams modified existing solutions that worked on real-world applications to compete in the CGC, and other teams took the CGC as a chance to implement new or existing techniques to test their utility. Since the Shellphish team built their Driller solution on top of `angr` and AFL the transition of Driller from CGC to

[21] `https://yurichev.com/tmp/SAT_SMT_DRAFT.pdf`
[22] `https://triton.quarkslab.com/files/csaw2016-sos-rthomas-jsalwan.pdf`
[23] `https://0xec.blogspot.com/2016/04/reversing-petya-ransomware-with.html`

real-world binaries doesn't sound too difficult at the surface. However, we describe the challenges that appear in real operating systems, specifically Linux, compared to the CGC.

2.3.1 Challenges

The main challenge of transitioning from a CGC solution to Linux hinges on the fact that these automated solutions are to perform a *full* analysis of the test programs. This means starting from the beginning of the program and ending either when it exits normally or crashes for a specific input. As dynamic analysis tools trace input data through a program, handling the side-effects and complete functionality of an operating system and execution environment presents many challenges. We describe three important challenges that a tool such as Driller must solve to be viable for real-world vulnerability analysis:

1. Nondeterminism
2. Library and System calls
3. Flexible File System

2.3.1.1 Nondeterminism

Nondeterminism can prevent the reproduction of a crash or hinder path exploration when searching for new inputs. There are multiple sources of nondeterminism; some are easier to fix than others. Among the easiest to fix are functions that use a seed for randomized value generation or functions that are commonly used to generate seeds for random-value functions, i.e. the C library functions `srand`, `rand`, `time`, etc. These function calls can be intercepted to log random values produced, or the output values can be symbolized and and unconstrained. Consider code piece 2.2. On line 5, there is a 5% (1/20) chance that the expression will be true and `logic_A` will execute. Without intercepting the `rand` function and returning an unconstrained symbolic variable, we would have to wait until a random value that satisfied this constraint was produced before we could explore `logic_A`, and then our generated inputs will not be reproducible unless that same random value was generated again.

Listing 2.2 : Random Number Example.

```
time_t t;
/* Intializes random number generator */
srand((unsigned) time(&t));

if (rand() < RAND_MAX/20) {
  <logic_A>;
}
else {
  <logic_B>;
}
```

Other, more difficult, sources of nondeterminism that resist test-case reproduction include race conditions that are architected within the program like threading, background processes that overload the CPU, differences between versions of operating systems, hard drive read errors, etc. We do not focus on these in this paper as they are out of scope, and this can be examined in future research.

2.3.1.2 Library and System Calls

Almost all production software makes use of existing code libraries and system calls. Library and system calls perform many common operations such as `strlen`, returning string length, `printf`, printing a formatted string, `malloc`, allocating memory, `read`, to read from a file, etc. Often, the objective is not to look for vulnerabilities in these library functions, but to examine how the program uses their results. Thus, it is important to intercept, or *hook*, these known library and system calls to prevent superfluous and wasteful analysis time during concolic execution. Section 3.2.2 goes into more detail about how we handle hooked library functions.

Not only is it challenging to correctly reproduce the logic found in library and system calls, but these functions can also interact with the operating system and execution environment. For instance, the C library call `setenv` will set a specified environment variable with a value. This environment variable can then be used by other functions like `getenv` which will return the value of the specified environment variable. If we are to replace these library functions, our solution must implement some type of persistent memory bank to store, retrieve, and update environment variables.

Another challenge is deciding which version of a library call to implement if the logic of the call changes drastically between versions. This also means that there is a maintenance challenge if a library function replacement implementation is to be made publicly available and robust.

2.3.1.3 File System

The execution environment where the CGC challenges ran did not support file persistence. One of the big challenges is simulating a file system that can maintain accurate persistence throughout analysis and control flow exploration. Integrating a robust file system is important in applying our analysis methods to the broad set of programs that use and interact with files during their execution.

2.3.2 Existing Solutions

Almost all existing vulnerability analysis tools have been developed for and tested on production software operating in real operating system environments. There are a number of papers that describe both static and dynamic tools that aim to analyze real-world binaries, with KLEE and S^2E being two of the most popular open source tools. The developers of these tools have had to overcome the obstacles that real execution environments present. We now lightly explore the technical methods used to overcome some of these obstacles during both static and dynamic analysis in order to better understand the existing base that our work builds off.

2.3.2.1 Static

Static analysis methods for discovering vulnerabilities and unsafe coding practices can be seen not only in compiler warnings or errors but also in modern IDEs like Microsoft's Visual Studio, Eclipse,[24] Pycharm,[25] etc. Furthermore, sophisticated algorithms can process source code to track how variables are initialized, modified, and accessed to find bugs like buffer overflows [25].

There are also commercial and research frameworks built for the static analysis of compiled programs. Plugins for IDA Pro[26] search for assembly code patterns as

[24] `https://eclipse.org/`
[25] `https://www.jetbrains.com/pycharm/`
[26] `https://www.hex-rays.com/products/ida/`

points of interest for potential exploits, like ROP gadgets [27]. IDA Pro provides a documented API to allow developers and researchers to write plugins and extend its features, however its core analysis functionality is closed-source. Another tool, `bap` (binary analysis platform), developed by researchers at Carnegie Mellon, can disassemble and statically analyze fully-featured programs to discover vulnerabilities [26]. Written in OCaml, `bap` is open source and actively maintained.[28]

2.3.2.2 Dynamic

Driller is different than most other established tools and techniques because it `combines` both fuzzing and concolic execution. As previously mentioned, fuzzers like AFL, Radamsa, boofuzz, Peach Fuzz, and others use heuristics to speed up the fuzzing process. These heuristics and techniques can range from binary instrumentation in AFL to formal grammars of a protocol's specification in Peach Fuzz. Notable symbolic execution engines like Triton, KLEE, S^2E, and `angr`, alone, are best suited for small portions of real programs due to the weaknesses of this analysis technique described in Section 2.2.2.2 like path explosion and semantic correctness. However, KLEE and S^2E have been evaluated against a set of real linux utility programs [9] with promising results.

Furthermore, there are commercial and consulting services offered by the likes of Microsoft and ForAllSecure—Project Springfield and Mayhem, respectively—that combine fuzzing and symbolic execution for increased effectiveness. Internally, Microsoft uses SAGE for testing its Windows operating system [27]. Based on the papers published about SAGE and Mayhem, the tools are effective on real-world, production software.

[27] `https://github.com/iphelix/ida-sploiter`
[28] `https://github.com/BinaryAnalysisPlatform/bap`

Chapter 3
Implementation and Methodology

3.1 Making the CGC Binaries More Realistic

Reproducing the vulnerability discovery results from the CGC binaries on a real system is not simple. The binaries were compiled for the 32 bit x86 instruction set and only used 7 system calls, with all other helper and library functions being written and statically compiled into the binary. Already, there is an existing effort to port the binaries for execution in a Linux environment, `cb-multios`,[1] but this project does not support the replacement of statically linked C library functions with the more commonly seen dynamically-linked C library functions from the system. Thus, in the interest of time, we make our best effort to modify the challenges to make use of the system's C library implementation and test correctness by ensuring that any crashing inputs generated by our tool will only crash an unpatched challenge and not a patched challenge. We pick the same subset of 13 binaries for which Driller was effective in [1] and modify these challenges to be more realistic. Table 3.1 lists these binaries by name, as found in the `cb-multios` project, and their respective challenge mappings as listed by the CGC.[2]

In this section, we elaborate on the following challenges we faced and how we decided to overcome them when modifying these challenge binaries:

- Compiling for 64-bit systems: Section 3.1.1

[1] `https://github.com/trailofbits/cb-multios`
[2] `https://github.com/CyberGrandChallenge/samples`

- Inserting and dynamically linking GNU standard C library functions: Section 3.1.2
- Limitations and correctness testing for challenge modifications: Section 3.1.3

The CGC binaries provide an invaluable dataset with documentation on the vulnerabilities within the applications. These challenges binaries are programs that provide real functionality and should not be dismissed as only being synthetic tests. Thus, a complete port of these binaries to a Linux environment would provide a great dataset to the vulnerability and program analysis community and provide a standardized dataset with which implementations and tools may be compared.

Chosen CGC Binaries

Audio_Visualizer	KPRCA_00010	FISHYXML	NRFIN_00030
Griswold	NRFIN_00017	HIGHCOO	NRFIN_00022
netstorage	KPRCA_00008	PCM_Message_decoder	CROMU_00004
PKK_Steganography	KPRCA_00012	SOLFEDGE	NRFIN_00020
stream_vm	KPRCA_00014	stream_vm2	KPRCA_00035
ValveChecks	NRFIN_00016	Vector_Graphics_Format	CROMU_00018
Vector_Graphics_2	CROMU_00036		

Table 3.1: The table of our chosen CGC binaries that we modify to be more realistic. Shown is the human-friendly name and the CGC identifier. These binaries are a subset of all CGC challenges and were picked based on the effectiveness of Driller in [1]

3.1.1 Compiling for 64-bit Systems

Today, modern desktop computers use a 64-bit operating system. However, the CGC binaries were built for the 32-bit DECREE execution platform. The analysis of 32-bit and 64-bit binaries is similar, but there are subtleties and nuances between the two architectures that an automated analysis system must take into consideration. To name a few examples, the number of instructions, function calling conventions, and the size of registers, addressable memory, and some data types all differ between 32-bit and 64-bit compiled programs.

Regarding the CGC binary dataset, there are instances where code uses the return value of `sizeof(void *)`, which returns the size of a pointer, a value of 4 for 32-bit environments and 8 for 64-bit environments. We have not investigated

this, but an issue on Github has been raised.[3] For our 13 applications, we briefly reviewed the source code and did not find any obvious instances where a port to 64-bit architecture would affect results. However, porting software from 32-bit to 64-bit is not straight-forward. We leave a more thorough investigation to future work.

3.1.2 Linking C Library Functions

The DECREE system did not support binaries with dynamically linked libraries, which means that the developers of the challenge applications wrote their own implementation of C library functions. While some of the challenge application developers packaged their implementations into a single `libc.c` file that was compiled with the application, others separated their library functions into multiple files, each containing the required library functions, i.e., `malloc.c`, `memcpy.c`, `printf.c`. Thus, in order to replace these calls, we removed the respective `#include` header statement commands and included the appropriate system C library headers.

3.1.3 Limitations and Testing Correctness

The CGC binaries are reincarnations of real programs and real programs' functionality, modified to operate in DECREE. Therefore, they are non-trivial, and they require test cases to ensure correctness after modifications. The `cb-multios` project is currently the best effort at reproducing the CGC binaries in a real Linux environment. With 32-bit compilation and enabling test cases, the initiative has achieved reproducible proof-of-vulnerabilities (POVs) on 91.45% of all POVs for all challenges.[4] Thus, achieving full correctness, beyond what is provided by `cb-multios` and for 64-bit compilation, is out of the scope of this thesis.

We satisfy our dataset's correctness for vulnerability replication by ensuring that any crashing inputs generated with our modified 64-bit challenge binaries will also crash the binaries compiled by the `cb-multios` project.[5] Additionally, we remove any single-instance random number challenge-response functionality

[3] `https://github.com/trailofbits/cb-multios/issues/14`

[4] As of June 4, 2017, `https://docs.google.com/spreadsheets/d/1Z2pinCkOqe1exzpvFgwSG2wH3ZO9LP9VJk0bm_5jPe4/edit?usp=sharing`, lists all passing and not passing tests.

[5] Where appropriate, randomness has been removed for both modified and original challenges when testing correctness.

because we only focus on the introduction and handling of C library functions with our modifications and testing. Handling randomness can be addressed in a similar manner, by knowing the random seeds, as that found in the original Driller paper, [1].

3.2 Symbolic C Library Function Summaries

C library functions provide commonly used routines and constructs like string finding and file-related data structures. The library functions are used by programmers to speed up development of new applications and are trusted to perform correctly. When performing program analysis in most cases, we are only interested in how the analyzed program uses the *results* of the library functions; we do not want to analyze the implementation of the library function itself. The C library functions found in GNU's `glibc` contain esoteric system calls and coding or compiler constructs not typically found in user-land software. Thus, to mitigate path explosion incurred during symbolic execution of system calls and C library functions, many tools like KLEE and S^2E choose to execute a custom model or *summary* that performs the desired operation without path explosion. While KLEE and S2E choose to model just system calls, Driller and its underlying symbolic execution engine, `angr`, choose to summarize *both* system calls *and* C library functions.

In this section, we elaborate on the following implementation challenges and detail how symbolic C function summaries work in Driller:

- Challenges of processing symbolic data within a C library function summary: Section 3.2.1
- How a symbolic C function summary works: Section 3.2.2

The choice to accurately summarize C library functions by hand is certainly an enormous task and prone to error. However, these summaries go a long way in preventing path explosion when tracing through programs with linked C libraries. Even with only accurately modeling system calls, like S2E and KLEE choose to do, there are still over 300 system calls in Linux Kernel version 4.4.0-77 on a 64-bit Ubuntu 16.04 machine. Both KLEE and S2E have shown to be applicable in a variety of real application datasets, including Windows programs [10] and `coreutils` [9]. Thus, summarizing both system calls *and* C library functions

will speed up analysis and increase the effectiveness of this vulnerability analysis technique when applied to real systems and applications.

3.2.1 Challenges of Implementing Symbolic C Library Functions

One problem with summarizing these library functions is the correctness and complexity of the results. This is especially true when dealing with symbolic data in a function like `strstr(char *haystack, char *needle)`, which tries to find the `needle` string within the `haystack` string. If the needle and haystack are both symbolic, then there is a possibility that the data and length of both the needle and haystack are unbounded, which leads to complex value constraints and may even be unsolvable. Therefore, symbolic execution frameworks sometimes place limits on symbolic data in order to have a tractable solution, with the trade-off being a more limited real-world application. Fortunately, the existence of test cases from the `glibc` source code and community contributions help to resolve corner cases and accuracy issues as they are encountered during the development of function summaries.

Generating and solving constraint expressions on symbolic data is very resource intensive as the number of dependent symbolized bytes and constraints grows. Therefore, it is necessary to limit, reduce, or selectively ignore some symbolic bytes to improve constraint solving time and complexity. However, choosing these limits, methods of reducing constraint expressions, and techniques to selectively ignore symbolic bytes can all affect the veracity of the generated result [28].

3.2.2 How a Symbolic Function Summary Works

The goal of a function summary is to provide the same functionality as the original linked and compiled function but in a way that avoids the path explosion found in some native implementations of the functions. Briefly, `angr` utilizes a function summary by *hooking*, or intercepting, the external call or jump to the dynamically linked function in the program. Thus, instead of executing native compiled library code, a function summary performs the necessary operations and, when done, returns back to the native code, just like the native program expects. More information about function summaries (or `SimProcedures`, as they're called in the

implementation) can be found in the `angr` documentation.[6].

3.3 File and File System Implementation

Since the DECREE system did not support files or a file system and almost every real-world binaries makes use of files, it was paramount that support for a file system be added to Driller. While `angr` had basic support for opening, reading, and writing to files, it was apparent that more advanced and complete file system features were required during the testing of real Linux binaries. Specifically, Driller did not correctly support the multiple opening and closing of the same file. To support it, we implemented a more accurate representation of a file system. Our implementation is not complete, but it has passed the related C library function tests found in GNU's C library implementation, a reasonable benchmark for correctness and usability.

In this section, we elaborate on the following points:

- What were the existing capabilities of Driller and `angr` with respect to files? Section 3.3.1
- Adding support for multi-opening of the same file: Section 3.3.2
- Adding symbolic file content support to Driller tool: Section 3.3.3
- Notable missing, unimplemented, or subtle file-related features: Section 3.3.4

3.3.1 Existing File Handling Capabilities and Limitations

While Driller handled files in a simple manner, this base support was minimal. First, `angr` has an initialization option to respect the execution environment and load a concrete representation of the file or to use symbolic data for a file's content. Second, when a file was opened, it was loaded into a data structure that contained a mapping of the file's *file descriptor*, a unique integer identifying the open file, and a data structure holding the file's path, content, and other file-related metadata, as shown in Figure 3.1. Third, a program could read, write, and seek a file. However, if a program tried to close a file, `angr` would not release its file descriptor number,

[6] `https://docs.angr.io/docs/simprocedures.html`

which does not accurately support the opening and closing of the same file because a new file descriptor would be issued each time the file was open, even if it was the same file path.

Previous File System Design

Path	Metadata
/dev/stdin	R,fd=0,…
/dev/stdout	W,fd=1,…
/dev/stderr	W,fd=2,…
/tmp/test.txt	R/W,fd=3,…
/tmp/test.txt	R,fd=4,…
…	…

Figure 3.1: A simple graphical representation of the new file system implementation.

3.3.2 Multi-opening of the Same File

To support opening the same file multiple times, we had to implement correct file descriptor operation and file closing. We follow a simple file system and operating system model to support files as described in [29], Chapters 39 and 40. When a file is opened, it is given a unique file descriptor (positive integer) that the program can use to read, write, seek, and other operations. When the file is closed, the file descriptor is given up and another newly opened file is now the reference. Figure 3.2 shows a simple graphical representation of the underlying implementation.

As mentioned in the previous section (Section 3.3.1), `angr` did not implement file descriptors or file closing correctly: if the same file was opened, closed, and then opened again, none of the modifications were seen in the second opening of that file. Listing 3.1 shows an example program that illustrates how the old (Listing 3.1) and new (Figure 3.2) file handling designs operate. In the old system, the assertion made on line 17 would have failed because the old file descriptor (`3`) would not have been released for reuse after the `close` on line 12. Furthermore, the assertions on lines 18 and 19 would have failed because write would have never been seen due to the engine reading an entirely new file, even though it's actually the same file in reality. Our new file system implementation works on this example.

Listing 3.1 : File Multi-open Example.

```
1  ...
2  char *name;
3  FILE *fp = NULL;
4  int fd;
5  char buffer[256];
6
7  name = "/tmp/test.txt";
8  fd = open(name, O_RDWR);
9  printf("open fd: %d\n", fd);
10 assert (fd == 3);
11 write(fd, "foobar and baz", 14);
12 close(fd)
13
14 fp = fopen (name, "r");
15 printf("fopen fp: %d\n", fileno(fp));
16 assert (fp != NULL);
17 assert (fileno(fp) == 3);
18 assert (getc (fp) == 'f');
19 assert (getc (fp) == 'o');
20 ...
```

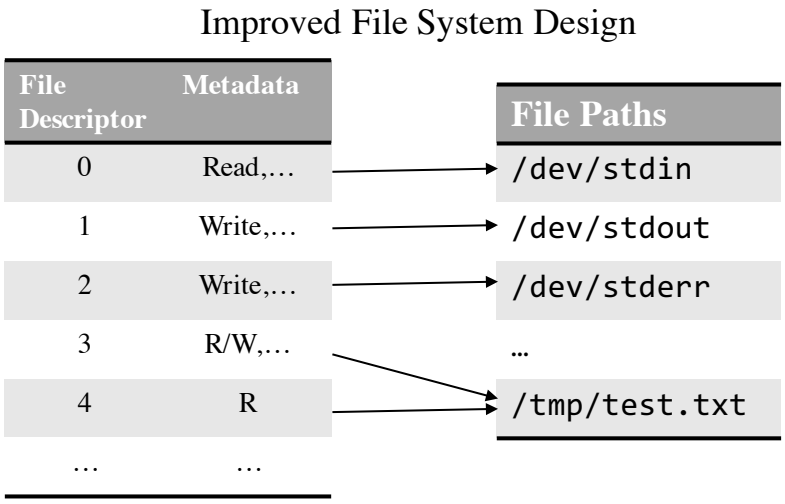

Figure 3.2: A simple graphical representation of the new file system implementation.

In order to complete these changes in the Driller and `angr` source code, we modified the `posix` object, added multiple new C library functions summaries, and utilized test programs from the GNU C Library implementation to test for correctness. We describe our modifications here.

`posix` modifications. The `posix` class is a plugin for `angr`'s simulation engine, SimuVEX,[7] and it handles "information about the operating system or environment model,"[8] including files. Keeping track of the file descriptor mappings and the file content was added here. This involves copying the state of the file system and file descriptors when a new path is found by Driller and removing the file descriptor when a file is closed.

Contributed function summaries. Performing analysis of programs that use files requires the support of the respective C library function summaries used by the program. Thus, in our exploration and experiments, we provided support for ten function summaries, or `SimProcedure`'s. This includes `fdopen`, `fputs`, `tmpnam`, `unlink`, `getenv`, `_IO_getc`, `fileno`, `ctype_toupper_loc`, `ctype_tolower_loc`, and `ctype_b_loc`.

Correctness testing of function summaries. Correctness testing is difficult for these function summaries due to the nature of symbolic variables. The fact that symbolic variables can take on any value within the constraints placed on them means that correctness testing must exercise both symbolic and concrete functionality for all function arguments. In this paper, we tested our implemented functions by selected tests from GNU's C Library, `glibc`.[9] While these tests are mainly concrete, our contributed functions mainly use concrete data or symbolic data that is not modified. We leave thorough symbolic data testing functionality to future work.

3.3.3 Symbolic File Content Support for Driller

Since Driller was only applied to the CGC binaries and those binaries only accepted input on `stdin` with no program command-line arguments, we modified Driller to accept command-line arguments and specify a symbolically-backed file that was under

[7] `https://github.com/angr/simuvex`
[8] `http://angr.io/api-doc/simuvex.html`
[9] `https://sourceware.org/glibc/wiki/Testing/Testsuite`

a user's control. For instance, in Section 4.3.1 we analyze the program executable `peinjector`. The program takes options on the command-line to provide different functionality; and in this case, we want to print out information about the file, signaled by `-info` on the command-line. `peinjector` takes a file path when paired with the `-info` option, so we pass it a file's path: `./peinjector --info notes.exe`. However, we don't want to load the concrete data of this file because then we wouldn't be able to find new paths during analysis, so we indicate that the file's content is to be symbolic by replacing the path with `@@`: `./peinjector --info @@`. The `@@` identifier was chosen based on the same usage in AFL, Driller's fuzzing component. The file's path does not matter, but if it did, AFL has an existing method to specify a file's path; but this extension is left as future work for Driller.

3.3.4 Missing Features

File systems and their different implementations include many features that are both commonly and rarely used. For our implementation, we do not claim to cover all features. We aim to cover the most important features that allow for an effective analysis of most applications. Therefore, we leave additional feature implementations to future work. We now list features we did not have time to implement or test, and those that will affect applications that utilize this functionality.

Reading content from multi-open files. When opening the same file twice, each file descriptor object will keep track of its own read/write/seek position in the file's contents. This becomes troublesome when writing to two different file descriptor objects pointing to the same file. In our implementation, we only synchronize new content to the backing file system when a file descriptor is closed with the `close` or `fclose` (and other similar) functions. This means that whichever file descriptor is closed last will overwrite the file's content on the file system. Utilizing the tests provided with GNU's C Library implementation will help with future testing of correct implementation techniques and logic.

File metadata. The `stat` family of functions return the status of a specified file. This includes the user ID of the file owner, group ID of the owner, total file size, etc.[10] While all of this data is important and used in applications, we do not choose

[10] `http://man7.org/linux/man-pages/man2/stat.2.html`

to implement or keep track of all of these fields, yet, and we leave this as future work.

File operation failure simulation. Many programs will execute alternative functions and routines if and when file operations fail. KLEE and S2E provide an option to have file operations fail by a given probability. Currently, `angr` focuses on operability and, by default, will give the program anything it wants. For instance, if the program tries to open and modify a file that is normally owned by the administrative user, then `angr` will let it. This becomes troublesome if the program uses that functionality as a specific check that it knows will always fail to run a different execution path and functionality, e.g., a potential obfuscation technique used by malware.

Chapter 4
Experimental Results

4.1 Introduction

In this chapter, we explore and test the efficacy and performance of Driller on the modified CGC binaries, compare Driller against other state-of-the-art open source implementations and techniques, and conduct an analysis of popular C library functions used in different categories of programs.

For our analyses, we use the `gcov` Linux utility program to obtain a code coverage metric for the test cases generated by each tool. Code coverage gives a measurement based on the number of *source code lines* executed by a set of inputs. This is a percentage of the total number of executable source code lines. While complete code coverage does not sufficiently indicate thorough testing and analysis, it is an important metric [30–32]. There are other granularities by which we could measure code coverage, i.e., basic blocks or instructions, however, since we have access to all dataset source code and because `gcov` is easy to use, it is our utility of choice for this measurement.

Furthermore, the analysis of popular C library function usage/frequency indicates which functions, out of the hundreds already implemented in Driller, are used by programs and worth spending more time in testing accuracy and completeness when summarizing by hand. We also group the unsupported functions by their respective header files to better reason about which sets of programs use what functionality.

4.1.1 Test Environment

While the CGC used a computing cluster consisting of 1280 CPU cores, 16TB of RAM and 128TB of storage.[1] For our experiments, we use a single Ubuntu 16.04 desktop machine running Intel Xeon CPU with 12 cores, 64GB RAM, and a ramdisk for fuzzing. We build our modifications on top of a snapshot of Driller and all related `angr` components from April 10, 2017.

4.2 Reproducing CGC Results

After replacing the CGC binaries with library routines, we test Driller using symbolized input at `stdin`. As mentioned in Section 3.1, we only test the 13 CGC binaries that utilized the advantages of Driller. We collect results for code coverage and whether we find a crashing input or vulnerability. We then compare Driller's performance against KLEE and AFL in Section 4.2.4.

4.2.1 Driller Results

Our Driller setup utilized 4 instances of AFL and a pool of 4 selective symbolic execution workers running `angr`. Each symbolic execution worker was limited to 16 GB of RAM and had a timeout of 2 hours before it would give up and test the next candidate input from the fuzzer. We let each binary run for 24 hours. We ran Driller with an initial seed that was 224 bytes long made up of random/nonsense words and characters. This length was chosen at random, but it provides a good starting point for the fuzzer and symbolic execution phase to explore the applications.

Of the 6 additional vulnerabilities found in the original Driller paper [1], we were able to reproduce 5. We first present the measurements of code coverage, C library function call frequency, speed, and found vulnerabilities for these 5 and then explain the difficulties and challenges encountered in reproducing these results.

4.2.1.1 Code Coverage

Table 4.1 shows the coverage results for each C source code file for the the modified CGC Griswold binary before a crash was found. A crash was found after 3.8 hours.

[1] `http://cs.ucsb.edu/~antoniob/files/hitcon_2015_public.pdf`

We collected similar measurements for the other CGC binaries in our dataset, but do not present them here. Table 4.3 shows the summarized code coverage percentages for the other CGC binaries as covered by Driller.

Coverage Metrics for Modified CGC Griswold Binary

`assemble.c`	38.44%	359 lines	`components.c`	33.61%	122 lines
`examine.c`	72.46%	69 lines	`operation.c`	94.29%	70 lines
`service.c`	100%	15 lines			

Table 4.1: This table shows the coverage metrics provided by GNU `gcov` after Driller found a crash on the modified CGC Griswold binary. Only the coverage metrics of the source code files for the binary are shown.

4.2.1.2 C Library Function Call Frequency

While the CGC binaries represent non-trivial programs, their functionality is heavily managed by developer-written code, even when replacing the standard C library functions. As Figure 4.1 shows, every binary makes use of basic memory library functions and data reads. This makes sense because there is no file system and these applications focus on mathematical and integer comparisons. Fortunately, the researchers and developers working on `angr` already implemented all of these C library functions;[2] and based on our ability to reproduce crashing inputs, these function summaries are accurate enough to be useful.

4.2.2 Speed Comparison

On average, the symbolic execution phase took between 200 and 300 seconds to trace or find new inputs for this dataset. Figure 4.3 shows the averages for each binary's time measurements in our modified CGC dataset. For the binaries which we found crashing inputs, we immediately halt analysis; and for those we did not, we let run until no new inputs are produced or 24 hours has elapsed.

The time taken to find the discovered vulnerabilities is dependent on the hardware available. Since we can distribute the work of fuzzing to multiple jobs on the same machine and multiple machines in a cluster, we can potentially improve the speed of fuzzing results by adding additional hardware. Faster processors,

[2] `https://github.com/angr/simuvex/tree/master/simuvex/procedures`

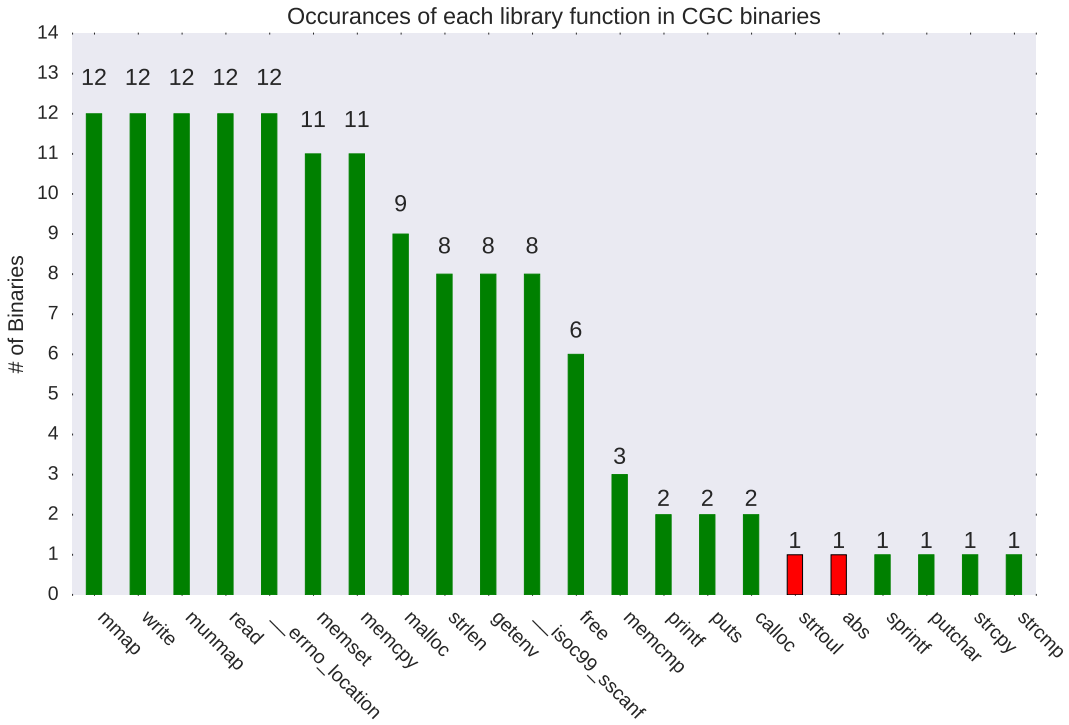

Figure 4.1: List of C functions replaced and the number of binaries they appeared in. The green bars indicate that the C library function summaries are available and implemented in Driller. Red bars indicate that no function summary exists yet for Driller.

additional RAM, and symbolic tracing efficiency improvements will all improve the speed and efficacy of the selective symbolic execution phase.

One improvement that was not utilized here, but is present in the `angr` project, is the use of a concrete emulation engine. Specifically, there is experimental support for utilizing the Unicorn[3] emulation project to speed up analysis of code areas which do not touch symbolized data. This would help with the execution of initialization code before any symbolic input was read.

4.2.3 Vulnerabilities Found and Challenges

Of the 13 binaries that we tested, we were able to find 5 vulnerabilities, we were able to invoke the DSE phase to gain additional coverage for 5, and the remaining 4

[3]http://www.unicorn-engine.org/

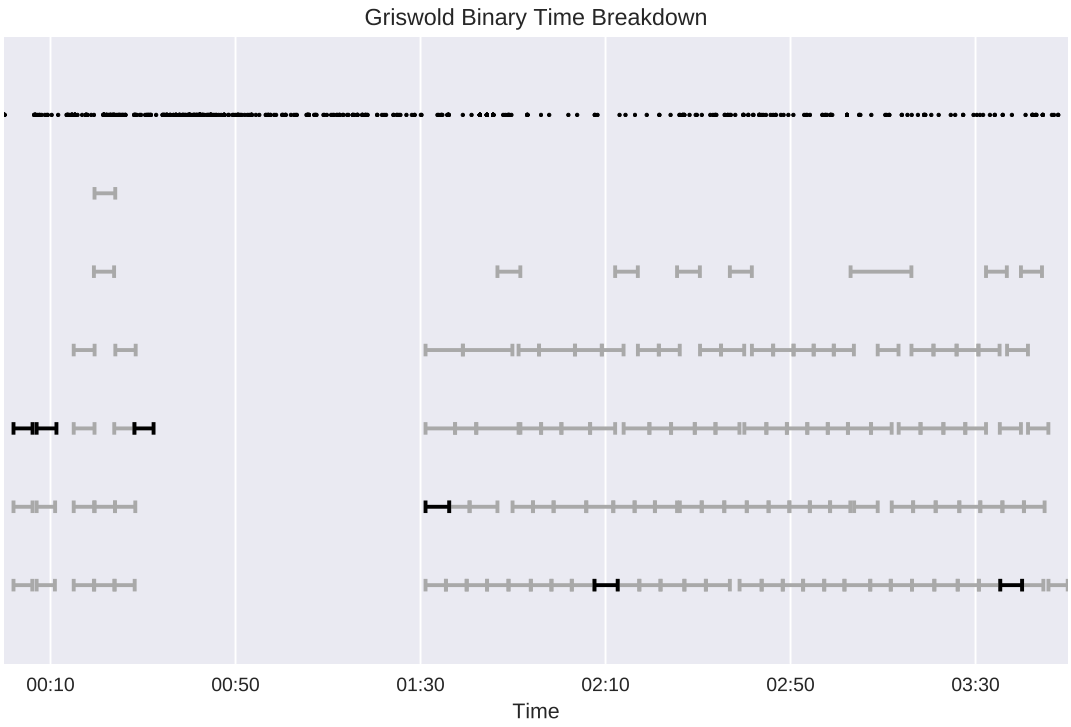

Figure 4.2: This plot shows the generation of test cases over time. The top row consists of a scatter plot marking the times when the fuzzer produced new and interesting inputs. The interval lines depict the start, stop, and duration of the DSE phase. Darker intervals indicate that new inputs were produced while the lighter intervals could not generate new inputs.

require additional support features added, as discussed in Section 4.2.3.1. Table 4.2 shows which binaries had crashes and which benefited from Driller's DSE phase. The reason we were unable to produce the same results as the Driller paper was due to a multitude of reasons:

- **Our modifications broke functionality.** Some of the custom replacement C library functions written by the CGC developers used alternative input-output mechanics. For instance, a noted difference was found in the `strcpy` function used by some of the binaries. Instead of returning the string copied, it returns the number of bytes copied. We tried to catch these non-standard functions, but a more robust testing methodology should be used in the future to ensure correct binary functionality after library function replacement.

- **C Library function summaries are inaccurate, incomplete, or oth-**

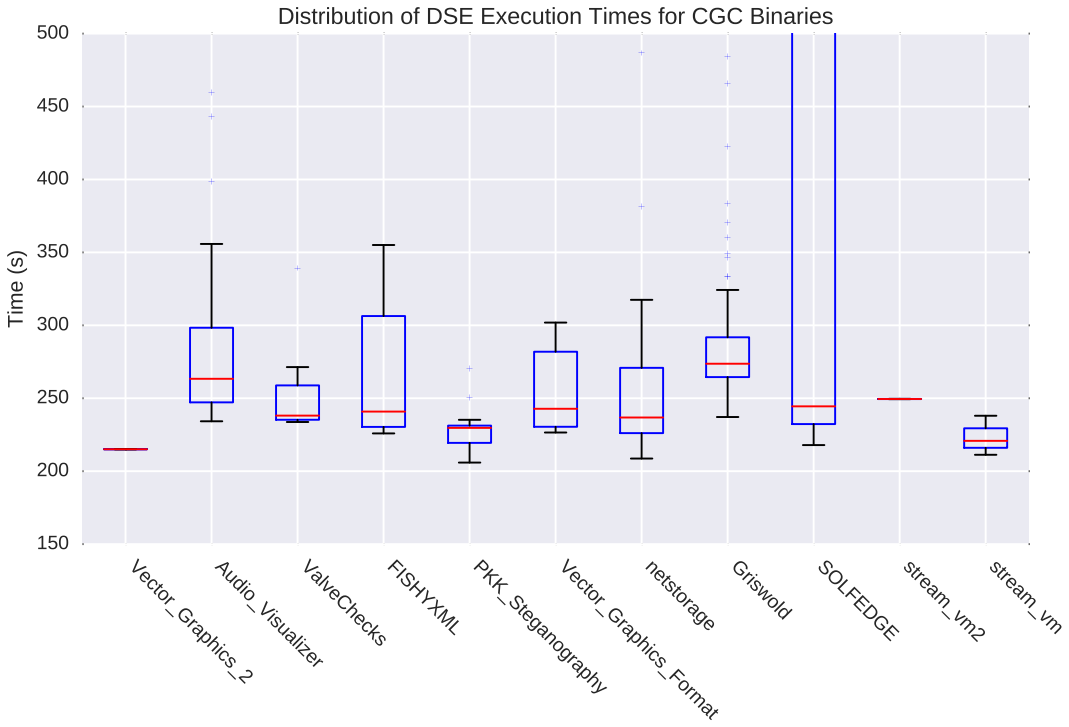

Figure 4.3: This graph shows the average times taken to generate new interesting inputs for each modified challenge. Much of the time spent in the DSE phase is due to filling the flag page memory space with random bytes.

erwise limited. Another reason for our inability to reproduce the Driller paper's results exactly is that the C library function summaries that were utilized during execution may not have been entirely complete. Much more time can be focused on testing the robustness of the 7 system calls' summaries for the CGC, while the C library functions are mostly built up and features added as needed.

- **Architecture dependent assumptions.** The CGC ran 32-bit binaries and the compilation to 64-bit instructions could have had some subtle yet incompatible effects. For instance, indexing into C `structs` with hard-coded indices could have influenced program behavior since pointer sizes and some data type sizes differ between 32-bit and 64-bit system architectures.

A more detailed description of our modifications and reasoning behind them is described in Section 3.1.

App. Name	Vuln. Found	DSE Cov.
Audio_Visualizer	✓	✓
FISHYXML	✗	✓
Griswold	✓	✓
HIGHCOO	✗∗ rand unsupported	✗∗ rand unsupported
netstorage	✗	✓
PCM_Message_Decoder	✗	✗∗ infinite loop
PKK_Steganography	✗	✓
SOLFEDGE	✗	✓
stream_vm	✓	✓
stream_vm2	✗	✗∗ Couldn't sat. new paths
ValveChecks	✓	✓
Vector_Graphics_2	✗	✗∗ mmap size too big for DSE
Vector_Graphics_Format	✓	✓

Table 4.2: This table shows a listing of the CGC binaries we tested and whether we found a vulnerability, marked in the *Vuln. Found* column. If we were unable to find a vulnerability, we mark whether the DSE phase discovered additional inputs that provided more code coverage in the *DSE Cov.* column. A '∗' represents a special note or challenge we encountered during analysis and is explained in Section 4.2.3.1

4.2.3.1 Special Notes

Due to time, we were unable to test or provide support for the **HIGHCOO** binary because randomness was a large part of the binary's functionality. An infinite loop occurred in the **PCM Message Decoder** binary and timed out after two hours of symbolic execution before Driller could find any new satisfiable paths. Driller was unable to satisfy any new paths for the **stream vm2** binary, and we are unsure why this happened as no critical errors or warnings were reported. Furthermore, after fuzzing the **Vector Graphics 2** binary, Driller used inputs that contained too large a value for `mmap`'s symbolic function summary and thus wasn't able to symbolically execute accurately.

4.2.4 Driller Comparison with KLEE

KLEE was unable to run the binaries after only modifying the source code to utilize `libc` functions. The flag page address mapping logic was unable to allocate the required number of bytes to the specific flag address. This meant that the

allocation of the flag bytes was patched out. Additionally, KLEE requires the inclusion of `(int argc, char **argv)` in the `main` function instead of empty or `void` arguments. Since KLEE only works on LLVM bytecode, which is mainly generated from source code, there is no support for custom, inlined assembly code.

4.2.4.1 Code Coverage Comparison

For this experiment, we use GNU `gcov` to obtain a line coverage measurement to compare Driller and KLEE. We find that Driller performs very well against KLEE for most binaries. Table 4.3 shows the coverage measurement comparisons against KLEE with an input size of 50 and 500 bytes. We use different input sizes with KLEE because KLEE has no way of extending the input size unlike Driller, which utilizes AFL's concatenation and combination modification techniques to increase test case sizes and cover more code.

While `gcov` is a popular code coverage tool, we had difficulties in obtaining measurements for CGC programs that were caught in an infinite loop with some of the replayed inputs. KLEE has its own tool to replay its generated test cases, `klee-replay`, which helped to resolve issues in that case, but for Driller, we were unable to have `gcov` exit the program gracefully and record the coverage results for **stream_vm** and **Audio_Visualization**. As mentioned before, we were unable to collect results for the **HIGHCOO** binary due to randomness.

We choose not to list AFL's code coverage here because it is a proper subset of Driller's code coverage. Furthermore, as discussed in Section 2.2.1.2, a fuzzer's effectiveness depends largely on the quality of the initial seed inputs.

4.2.4.2 Speed Comparison

A speed comparison between the different techniques may be useful in some cases, and even though we can report on the time taken by each implementation, the acceptable time to find a vulnerability is different for each scenario. On one hand, a CTF event may require contestants to find vulnerabilities quickly before an opposing team, and on the other hand, a state-sponsored vulnerability mining campaign may find days or weeks as an acceptable time-frame. With that in mind, Section 2.2.1.2 discusses the strengths of fuzzing and how a unique *seed* dataset will help with overcoming the complex-check limitations that fuzzers are unable

Binary	LoC	Driller	K-50	K-500	Diff.
Audio_Vis...	N/A	N/A	N/A	N/A	N/A
FISHYXML	869	26.01%	5.41%	5.41%	+20.6%
Griswold	635	48.82%	42.68%	42.68%	+6.14%
HIGHCOO	N/A	N/A	N/A	N/A	N/A
netstorage	246	22.67%	28.05%	58.54%	–5.38%
PCM...Decoder	253	23.32%	38.74%	19.37%	–15.42%
PKK_Steg...	232	52.16%	12.50%	12.50%	+39.66%
SOLFEDGE	197	97.97%	5.58%	5.58%	+92.39%
stream_vm	N/A	N/A	N/A	N/A	N/A
stream_vm2	157	18.47%	91.08%	42.68%	–72.61%
ValveChecks	172	38.95%	38.95%	38.95%	0.0%
Vector...2	431	5.80%	2.55%	2.55%	+3.25%
Vector...Format	539	89.98%	59.93%	60.30%	+30.05%

Table 4.3: A comparison of coverage measurements as percentage of lines of code (LoC) covered by Driller, KLEE with an input of size 50 (K-50), and KLEE with an input of size 500. The *Diff.* column shows the difference in code coverage between Driller and the highest percentage achieved by KLEE.

to solve. Furthermore, a distributed fuzzing campaign can speed up progress and results.

Symbolic execution, while sequentially executed during each run, can still be distributed for each new input. Similar to fuzzing, the acceptable time-frame for constraint solving is also dependent on the scenario and available resources. Enhancements to the internal representation of symbolic expressions, constraint solver heuristics, symbolic expression minimization techniques, etc. will all help to increase the performance and lower the time required to solve or find vulnerabilities.

With respect to the speed comparison for the modified CGC binaries, we find that a fuzzer with a random seed is soon unable to progress past complex checks and becomes stuck quickly. This was the expected result since we only chose the CGC binaries from the Driller paper where Driller was utilized. However, we found that KLEE found many of the same vulnerabilities as Driller and did so much faster. This is because the fuzzing phase in Driller takes longer to explore an entire compartment compared to KLEE, which can easily pass through complex checks to explore deeper within the binary.

4.2.4.3 Vulnerabilities Found

Of the 13 modified CGC binaries, Driller found the most vulnerabilities. Table 4.4 shows the vulnerabilities found by Driller versus those found by KLEE. Driller found 2 more vulnerabilities than KLEE in this dataset. This is different than the results presented in Figure 5 of the Driller paper [1] where Driller found 6 additional vulnerabilities than that of the combined results of `angr`, the tested symbolic execution engine, and AFL, the tested fuzzer, alone. Our results show that KLEE performs better than `angr` when used alone, but KLEE does not find more vulnerabilities than Driller.

App. Name	Driller	KLEE
Audio_Visualizer	✓	✓
FISHYXML	✗	✗
Griswold	✓	✗
HIGHCOO	N/A	N/A
netstorage	✗	✗
PCM_Message_Decoder	✗	✗
PKK_Steganography	✗	✗
SOLFEDGE	✗	✗
stream_vm	✓	✗
stream_vm2	✗	✗
ValveChecks	✓	✓
Vector_Graphics_2	✗	✗
Vector_Graphics_Format	✓	✓

Table 4.4: A comparison of the found vulnerabilities by Driller and KLEE for the modified CGC binary dataset.

4.3 Linux Binary Testing

Finding a good candidate binary was difficult given the lack of support for many C library functions. While the lack of C library support was detrimental in the search for finding new vulnerabilities, one must also consider the strengths of fuzzing: Of the 126 CGC binaries tested in the Driller paper, Driller only invoked the DSE phase on 13, and fuzzing proved to be sufficient for all others. Additionally, many modern open source projects utilize and take advantage of C++with its own

C++Standard Library in addition to the plain C Standard Library.

4.3.1 Case Study: PEInjector

PEInjector is an open-source project on Github[4] with over 400 stars (indicating popularity in the community). It operates on the Windows PE executable file format,[5] and "PEInjector provides different ways to infect these files with custom payloads without changing the original functionality. It creates patches, which are then applied seamlessly during file transfer." However, parsing the PE file format is complex with many acceptable corner cases.[6] Therefore, PEInjector's `--info` functionality, which parses and shows information about PE files, was the perfect candidate to test Driller's capabilities.

PEInjector reads the PE's filename from the command line, so Driller needed to utilize our added file handling support.

4.3.1.1 New Vulnerabilities

Driller found a vulnerability after 31 minutes. KLEE found the same vulnerability within a minute, but only when the input was large enough to contain a valid PE file.[7] AFL became stuck quickly when given an initial random input of size 224 bytes, but when given a simple valid PE file as an initial seed, AFL also found the same vulnerability within a minute.

The found vulnerability was a common read-out-bounds (CWE-125) vulnerability that occurred in a length/size field of the PE file. The parsing code looks to be written by the author of the tool and not taken from a well-tested or mature PE parsing library implementation. The custom parsing code did not check the field to ensure it was within the PE file's total file size. While PEInjector is likely not being used in a commercial environment or a scenario that requires constant availability, this vulnerability demonstrates the importance of using well-tested libraries to perform tasks such as file format parsing.

[4] `https://github.com/JonDoNym/peinjector`

[5] `https://msdn.microsoft.com/en-us/library/ms809762.aspx?f=255&MSPPError=-2147217396`

[6] `https://github.com/corkami/pocs/tree/master/PE`

[7] A valid PE file, according to PEInjector's implementation is one that is over 120 bytes, as taken from the PE32 file format standard for executable 32-bit programs.

It is important to note that although KLEE and AFL both found the vulnerability much faster than Driller, KLEE only works with source code, and AFL requires intelligent seeds. Therefore, when valid inputs are unknown Driller is able to generate valid inputs (test cases) that will cover more of the applications code.

Binary	**LoC**	**Driller**	**K-50**	**K-500**	**Diff.**
peinjector	485	37.94%	8.45%	10.31%	+27.63%

Table 4.5: Coverage comparison of Driller and KLEE with size 50 byte input (K-50) and size 500 byte input (K-500) as a percentage of lines of executable code (LoC).

4.4 Most Popular C Library Functions

To help focus on the most popular C Library functions, we look at a common collection of Linux utilities: `binutils` and `coreutils`. The programs in these collections represent many of the most common programs that users need to perform operations on their desktop. We look at each collection to determine the trends in functionality with respect to memory, files, permissions, and strings. With these results, we can focus on the correct implementation of C library function summaries and operating system feature-set for the targeted applications. Table 4.6 shows some common C Library headers and a description of the functionality provided.

4.4.1 `binutils`

We look at the `binutils` collection which contains 16 unique binaries and "their main reason for existence is to give the GNU system (and GNU/Linux) the facility to compile and link programs."[8] This collection of programs uses many string manipulation functions. We can see the number of supported vs. unsupported library functions in Figure 4.4. In order to fully support similar programs such as this, we can look at Figure 4.6, which shows the distribution of the number of programs that use functions belonging to each respective header. We find that, in addition to `<unistd.h>` having the most unsupported functions, string manipulation functions provided by `<string.h>` are unsupported as well.

[8] `https://www.gnu.org/software/binutils/`

Header	Description
<locale.h>	Defines localization functions.
<math.h>	Defines common mathematical functions.
<signal.h>	Defines signal handling functions.
<stat.h>	Defines the structure of the data returned by the functions `fstat()`, `lstat()`, and `stat()`.
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stddef.h>	Defines several useful types and macros.
<stdio.h>	Defines core input and output functions
<stdlib.h>	Defines numeric conversion functions, pseudo-random numbers generation functions, memory allocation, process control functions
<string.h>	Defines string handling functions.
<time.h>	Defines date and time handling functions
<uchar.h>	Types and functions for manipulating Unicode characters.
<unistd.h>	Defines miscellaneous symbolic constants and types, and declares miscellaneous functions.
<wchar.h>	Defines wide string handling functions.
<wctype.h>	Defines set of functions used to classify wide characters by their types or to convert between upper and lower case

Table 4.6: Popular C header files and a description of functionality.

4.4.2 `coreutils`

We look at the `coreutils` collection which contains 105 unique executable binaries that perform tasks related to "basic file, shell and text manipulation" and are "expected to exist on every operating system."[9] This collection of programs uses of the input output functions in `<stdio.h>`. We can see the number of supported vs. unsupported library functions in Figure 4.4. In order to fully support similar programs such as this, we can look at Figure 4.6, which shows the distribution of the number of programs that use functions belonging to each respective header. We find that, in addition to `<stdio.h>` having the most unsupported functions, miscellaneous functions provided by `<unistd.h>` and `<stdlib.h>` are unsupported as well.

[9] `https://www.gnu.org/software/coreutils/coreutils.html`

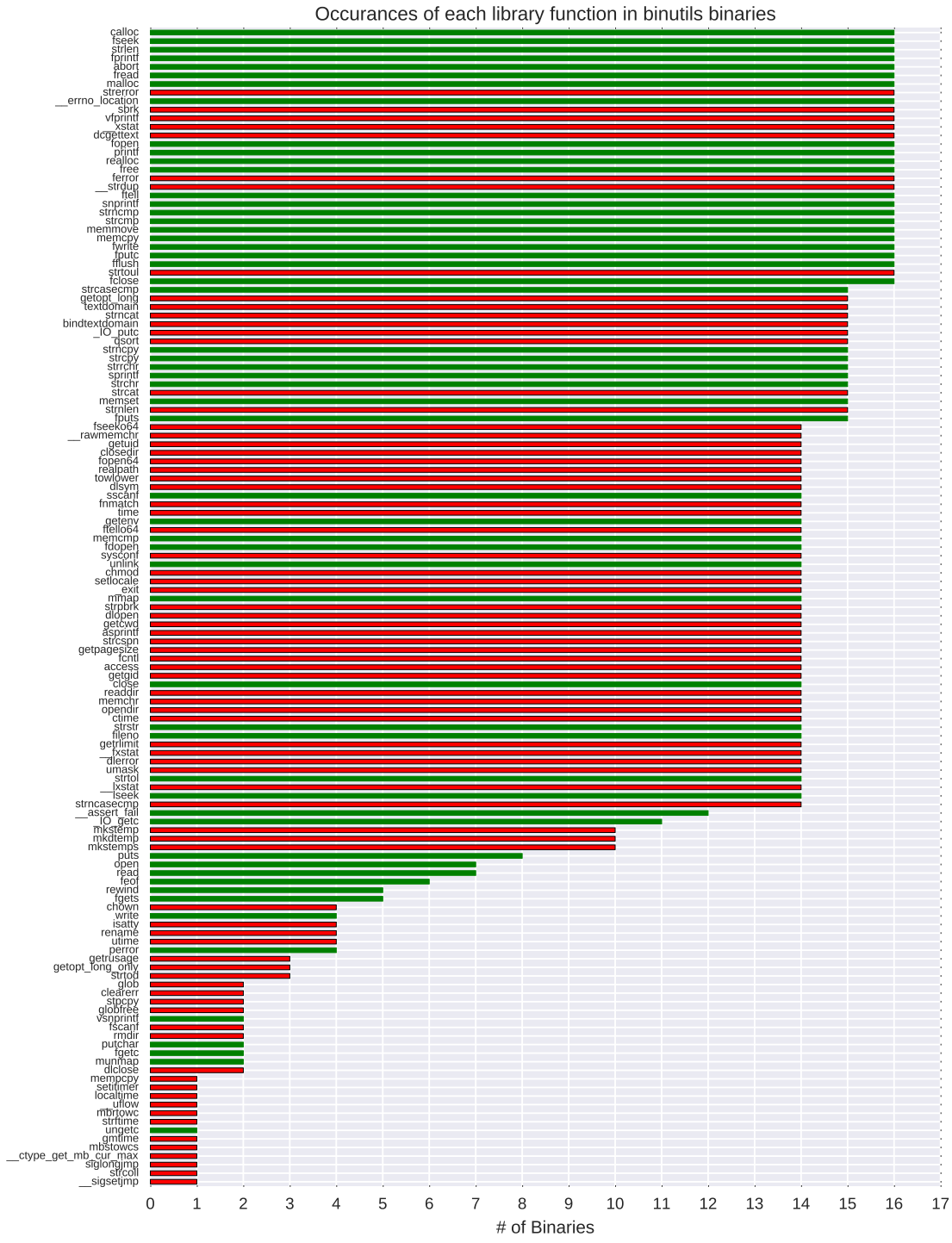

Figure 4.4: This shows the most commonly linked C library functions in the `binutils` v2.27 program set and whether or not they are supported in Driller. Red indicates no support, and green indicates at least basic support within Driller.

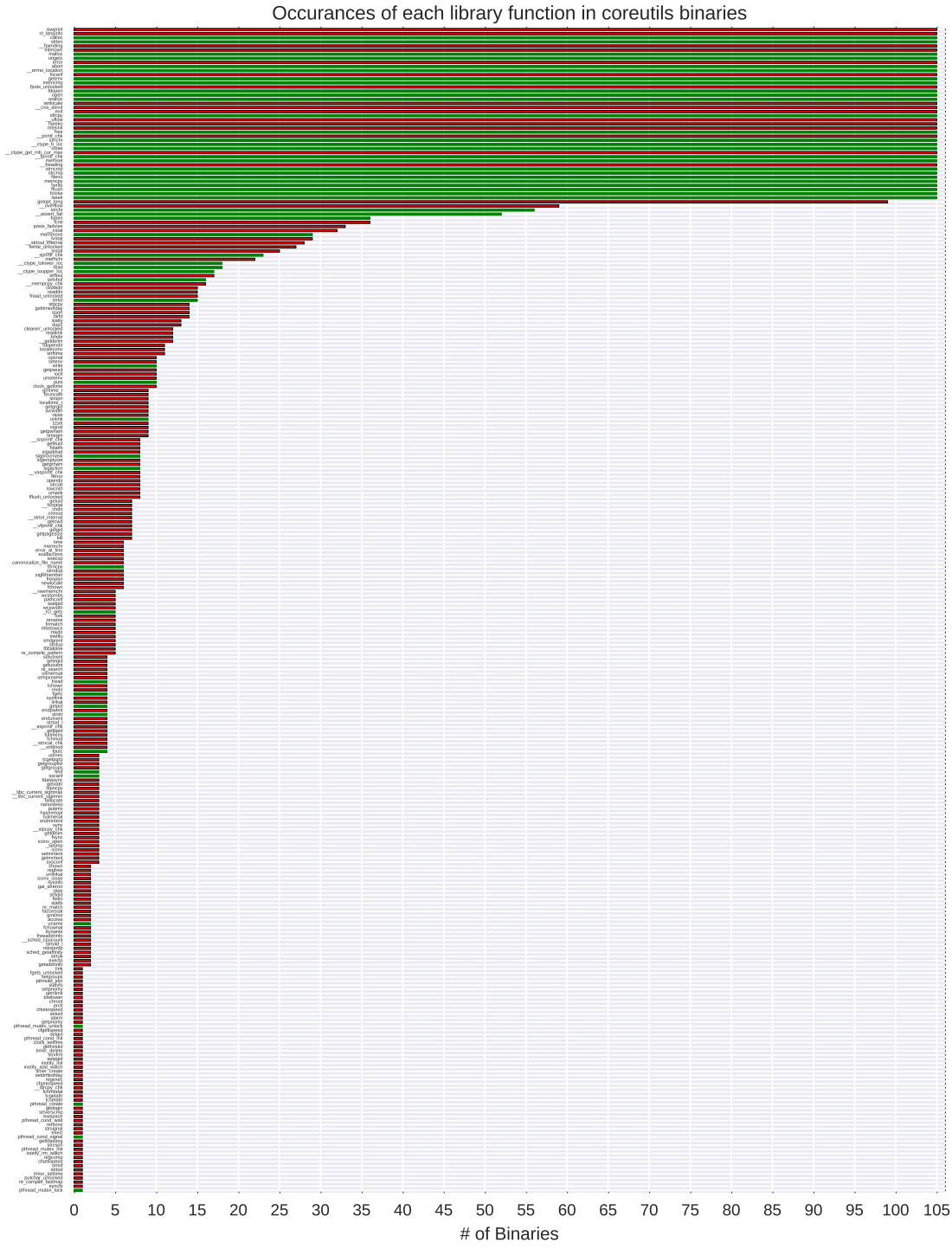

Figure 4.5: This shows the most commonly linked C library functions in the `coreutils` v8.26 program set and whether or not they are supported in Driller. Red indicates no support, and green indicates at least basic support within Driller.

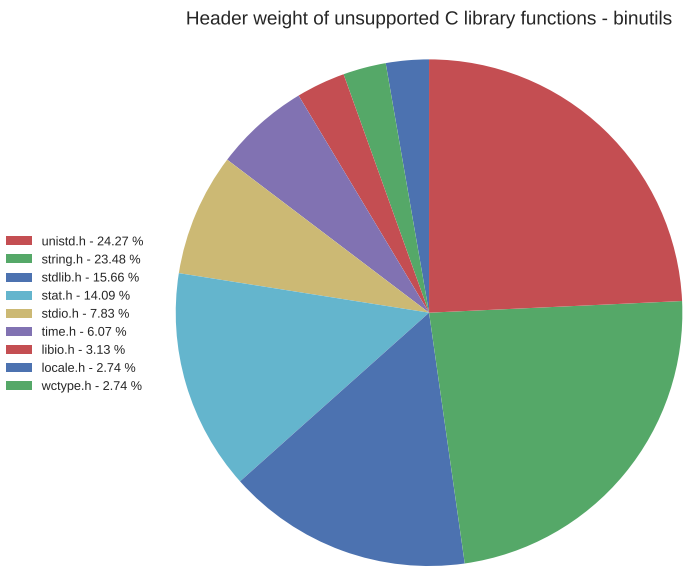

Figure 4.6: This shows the distribution of unsupported functions within the included C headers of the `binutils` v2.27 program set.

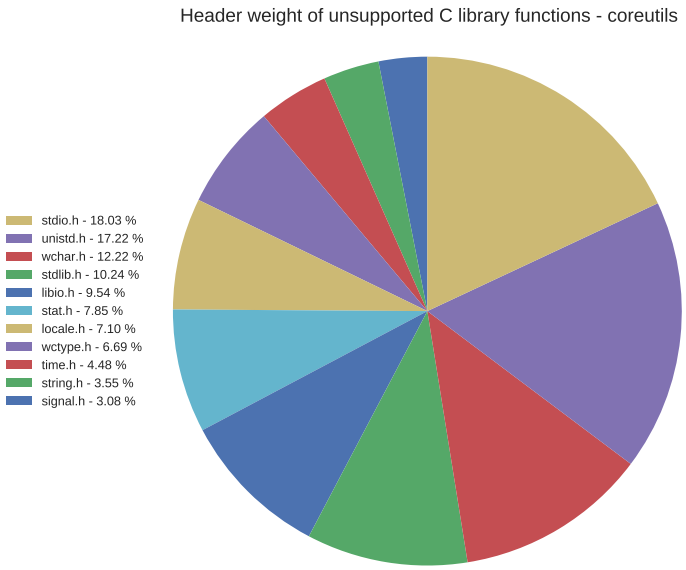

Figure 4.7: This shows the distribution of unsupported functions within the included C headers of the `coreutils` v8.26 program set.

Chapter 5
Conclusion

5.1 Limitations

The extensions to Driller and the `angr` framework completed during this research have been shown to work effectively for the chosen programs in our datasets. However, it is important to note that there are many caveats and limitations if applied to any arbitrary program. These limitations include a variety of fundamental limitations in our chosen constraint solvers, fuzzers, intermediate representation lifting and instruction modeling, and complex operating system functionality that are out of the scope of this paper. With respect to our goal in summarizing file-related and other C library functions, we have chosen to model the C run-time in the most simplistic way possible while still being effective in practice.

5.1.1 Program Property Limitations

There are certain program properties that make the effective use of Driller difficult or impossible with its current implementation. In general, Driller is most effective against binaries with memory comparisons or symbolic data with inter-dependencies and specific constraints, as these are the strengths of symbolic execution. In this section, we discuss a few program properties that are currently difficult to handle and how they may be solved in the future.

Scale. Even though fuzzing limits the effects of path explosion compared to pure symbolic execution, programs that contain long execution paths can generate long symbolic constraint expressions that are difficult to solve in a reasonable amount of time. Currently, there are efforts in improving expression minimization techniques

through quantifier elimination [33], improving constraint solving time using a portfolio of constraint solvers [34], and applying constraint solvers to more realisic scenarios [35]. Furthermore, multi-processing and threading introduce subtleties for performing correct analysis, like race conditions, process scheduling, and resource sharing. Although the `angr` team has modeled some threading functions, this feature is still in development for correct and robust operation.

Third-party Libraries. The vast majority of real programs will take advantage of code libraries written by other developers. These third-party libraries are not linked with the program in order to save space on the program's size during distribution. Currently, Driller is unable to trace paths outside of the code contained within the application's executable file, so there are three options to remedy this: trace the external function executions, create simprocedures for each external function, or link the binary statically to include all libraries except `libc`. A study of the strengths, weaknesses, implementation methods, and techniques of each option are left as future work.

Multiple Symbolic Inputs. A symbolic execution engine, including `angr`, fundamentally supports multiple symbolic inputs. However, AFL only supports one source of fuzzed input, thus extending Driller to operate on multiple symbolic inputs would require finding a fuzzer that supports multiple fuzz sources or by keeping track of the concrete data generated by the symbolic execution engine and utilizing that when fuzzing. This is also left as future work.

Windows and Other Operating Systems. Linux is the primary development platform for `angr` and Driller. Thus, support for Windows and other operating systems is still in development. The challenges associated with support for another OS are great: function calling convention is different, there are different system calls and functionality, the executable format is different, program loading is different, etc. However, the C library remains relatively the same, with the exception of some functions like those found in the `windows.h` header file. Thus, supporting more different operating systems will take time.

Random Numbers. The `srand` and `rand` functions, together, generate a random number. Randomized program functionality is useful in many applications, however randomization is difficult to reproduce and fuzz without a solution that accounts for it. The CGC binaries make use of randomization, and the Driller paper describes

a solution through the use of known seeds for random number generation and a subsequent process for generating a patch that accounts for random numbers [1]. Unfortunately, we did not have time to test randomized program functionality on real applications; so we patched out these areas of code as described in Section 3.1.3. We leave testing of randomized program functionality to future work.

Malformed Code. The analysis of malformed or custom, inlined assembly code is unsupported as any type of custom calling convention, control flow, or execution is possible. Therefore, the analysis of these programs, like malware or obfuscated code, must be analyzed by hand and on a case by case basis. However, the primary use-case of Driller is vulnerability discovery, which doesn't usually apply for malware analysis. However, programs that have been obfuscated, with no source code available, do fall into Driller's use-case. For these obfuscated applications, automated deobfuscation techniques may be useful [36].

5.2 Future Work

5.2.1 Alternative Implementations

In addition to the future works mentioned throughout the paper, we now briefly present more future work ideas.

Driller is made up of three distinct parts: fuzzer, tracer, and symbolic execution engine. A future study would perform a comparison between the replacement of each part with different implementations. Not only will speed be a measurement of interest, but support for different binaries of varying complexities and system architectures will also generate interesting results.

5.2.2 Automated Synthesis of Library Functions

Currently, the biggest issue with Driller and `angr` is the lack of support for library functions and the handling of these unsupported functions. Since the current tracer (QEMU) is unable to trace the execution within a library function and guide the symbolic execution, path explosion and unneeded logic is executed by the symbolic execution engine. If there were some way to automatically generate function summaries, given a set of formal specifications, whether extracted from

current C library implementations or integrated ex post facto, then unsupported library functions would not be a problem. Program synthesis [37], "the [automated] construction of a computer program from given specifications," could be a viable solution to this issue. However, program synthesis comes with many limitations, which include scaling issues with large programs. We leave an exploration of this potential solution to future work.

5.3 Conclusions and Takeaway

The CGC provided a benchmark to test and compare new automated vulnerability analysis techniques in a simplified environment. This simplified environment allows for the speedy development and testing of new techniques without worrying about the many complexities of a real operating system. However, we have shown that operating system complexities are unavoidable and troublesome when analyzing real binaries with new techniques and implementations based only on the CGC.

In an attempt to extend the best vulnerability discovery tool used at the CGC, we modified a subset of CGC binaries to more closely resemble real Linux binaries operating on an x86 64-bit system with dynamically linked C Library functions. We found that Driller was able to find 5 of the 6 vulnerabilities in the modified CGC binaries compared to the original Driller paper and covered more code compared to KLEE and AFL, with `angr` being more resilient in the loading and concrete execution of binaries than KLEE. Furthermore, we implemented more robust and effective file handling and a file system to enable symbolic execution support for applications that use files. Lastly, we studied the impact of unsupported C library function summaries and measured the most common, and thus important, functions that appear in different collections of Linux utilities to help direct the development efforts and focus of particular function summaries.

Automated vulnerability analysis is only growing in importance. The CGC binary dataset, once completely ported to different operating systems and architectures, will provide a useful, standardized benchmark for comparisons and performance measurements of future vulnerability analysis techniques and implementations. Lastly, this work would not be possible without the Shellphish team opening up their source code to the public, therefore we have contributed back our modifications to improve the Driller and `angr` implementation.

Bibliography

[1] Stephens, N., J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna (2016) "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*.

[2] Shoshitaishvili, Y., R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna (2017), "Team Shellphish - Cyber Grand Shellphish," `http://phrack.org/papers/cyber_grand_shellphish.html`.

[3] ——— (2016) "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*.

[4] Godefroid, P., M. Y. Levin, D. A. Molnar, et al. (2008) "Automated Whitebox Fuzz Testing." in *NDSS*, vol. 8, pp. 151–166.

[5] Böttinger, K. and C. Eckert (2016) "DeepFuzz: Triggering Vulnerabilities Deeply Hidden in Binaries," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, Cham, Cham, pp. 25–34.

[6] Nighswander, T. (2016), "Unleashing the Mayhem CRS," `https://forallsecure.com/blog/2016/02/09/unleashing-mayhem/`.

[7] Wang, T., T. Wei, G. Gu, and W. Zou (2010) "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," *Security and privacy (SP)*, pp. 497–512.

[8] Chen, T., X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu (2013) "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, **29**(7), pp. 1758–1773.

[9] Cadar, C., D. Dunbar, D. R. Engler, et al. (2008) "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, pp. 209–224.

[10] Chipounov, V., V. Kuznetsov, G. Candea, V. Chipounov, V. Kuznetsov, G. Candea, V. Chipounov, V. Kuznetsov, G. Candea, V. Chipounov, V. Kuznetsov, and G. Candea (2012) "S2E: a platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, **47**(4), pp. 265–278.

[11] "Common Weakness Enumeration (CWE)," The MITRE Corporation. `http://cwe.mitre.org/`, accessed: 2017-06-01.

[12] "The Bugs Framework (BF)," NIST. `https://samate.nist.gov/BF/`, accessed: 2017-06-01.

[13] "Common Vulnerabilities and Exposures (CVE)," The MITRE Corporation. `https://cve.mitre.org/`, accessed: 2017-06-01.

[14] Bessey, A., K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler (2010) "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, **53**(2), pp. 66–75.

[15] Viega, J., J.-T. Bloch, Y. Kohno, and G. McGraw (2000) "ITS4: A static vulnerability scanner for C and C++ code," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*, IEEE, pp. 257–267.

[16] Livshits, V. B. and M. S. Lam (2005) "Finding Security Vulnerabilities in Java Applications with Static Analysis." in *Usenix Security*, vol. 2013.

[17] Balzarotti, D., M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna (2008) "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, IEEE, pp. 387–401.

[18] Clause, J., W. Li, and A. Orso (2007) "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM, pp. 196–206.

[19] "Technical "whitepaper" for afl-fuzz," American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, accessed: 2017-06-01.

[20] Duchene, F., S. Rawat, J.-L. Richier, and R. Groz (2014) "Kameleon-Fuzz: evolutionary fuzzing for black-box XSS detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, ACM, pp. 37–48.

[21] Godefroid, P. (2007) "Random Testing for Security: Blackbox vs. Whitebox Fuzzing," in *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, ACM, New York, NY, USA, pp. 1–1.

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

[23] Schwartz, E. J., T. Avgerinos, and D. Brumley (2010) "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*, IEEE, pp. 317–331.

[24] Barrett, C. W., R. Sebastiani, S. A. Seshia, and C. Tinelli (2009) "Satisfiability Modulo Theories." *Handbook of satisfiability*, **185**, pp. 825–885.

[25] Zitser, M., R. Lippmann, and T. Leek (2004) "Testing static analysis tools using exploitable buffer overflows from open source code," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, ACM, pp. 97–106.

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

[27] Godefroid, P., M. Y. Levin, and D. Molnar (2012) "SAGE: whitebox fuzzing for security testing," *Queue*, **10**(1), p. 20.

[28] Ramos, D. A. and D. R. Engler (2015) "Under-Constrained Symbolic Execution: Correctness Checking for Real Code." in *USENIX Security*, pp. 49–64.

[29] Arpaci-Dusseau, R. H. and A. C. Arpaci-Dusseau (2015) *Operating Systems: Three Easy Pieces*, 0.91 ed., Arpaci-Dusseau Books.

[30] Patton, R. (2001) *Software testing*, Sams publishing.

[31] Chen, M.-H., M. R. Lyu, and W. E. Wong (2001) "Effect of code coverage on software reliability measurement," *IEEE Transactions on Reliability*, **50**(2), pp. 165–170.

[32] Devadas, S., A. Ghosh, and K. Keutzer (1997) "An observability-based code coverage metric for functional simulation," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, IEEE Computer Society, pp. 418–425.

[33] Monniaux, D. (2008) "A quantifier elimination algorithm for linear real arithmetic," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 243–257.

[34] OâĂŹMahony, E., E. Hebrard, A. Holland, C. Nugent, and B. OâĂŹSullivan (2008) "Using case-based reasoning in an algorithm portfolio for constraint solving," in *Irish conference on artificial intelligence and cognitive science*, pp. 210–216.

[35] Nieuwenhuis, R. and A. Oliveras (2006) "On SAT modulo theories and optimization problems," in *International conference on theory and applications of satisfiability testing*, Springer, pp. 156–169.

[36] Debray, S. K., C. Collberg, J. H. Hartman, D. K. Lowenthal, and S. K. Debray (2016) "Automatic Deobfuscation and Reverse Engineering of Obfuscated Code," .

[37] Manna, Z. and R. Waldinger (1975) "Knowledge and Reasoning in Program Synthesis," *Artificial Intelligence*, **6**(2), pp. 175 – 208.