Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution

Brian S. Pak

CMU-CS-12-116

May 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Professor David Brumley, Advisor
Professor David Andersen, Faculty

Submitted in partial fulfillment of the requirements for the degree of Masters of Science.

Keywords: Software, Security, Program Testing, Hybrid Fuzzing, Input Generation

Abstract

Random mutational fuzz testing (*fuzzing*) and symbolic executions are program testing techniques that have been gaining popularity in the security research community. Fuzzing finds bugs in a target program by natively executing it with random inputs while monitoring the execution for abnormal behaviors such as crashes. While fuzzing may have a reputation of being able to explore deep into a program's state space *efficiently*, naïve fuzzers usually have limited code coverage for typical programs since unconstrained random inputs are unlikely to drive the execution down many different paths. In contrast, symbolic execution tests a program by treating the program's input as symbols and interpreting the program over such symbolic inputs. Although in theory symbolic execution is guaranteed to be *effective* in achieving code coverage if we explore all possible paths, this generally requires exponential resource and is thus not practical for many real-world programs.

This thesis presents our attempt to attain the best of both worlds by combining fuzzing with symbolic execution in a novel manner. Our technique, called *hybrid fuzzing*, first uses symbolic execution to discover frontier nodes that represent unique paths in the program. After collecting as many frontier nodes as possible under a user-specifiable resource constraint, it transits to fuzz the program with *preconditioned random inputs*, which are provably random inputs that respect the path predicate leading to each frontier node. Our current implementation supports programs with linear path predicates and can automatically generate preconditioned random inputs from a polytope model of the input space extracted from *binaries*. These preconditioned random inputs can then be used with any fuzzer. Experiments show that our implementation is efficient in both time and space, and the inputs generated by it are able to gain extra breadth *and* depth over previous approaches.

Acknowledgements

I am thankful to have really awesome people around me who have helped me get this thesis out. There are too many people who have influenced me and my thesis to enumerate, but I do want to specially thank some of them explicitly.

First, I would like to thank my advisor Dr. David Brumley and thesis committee Dr. David Andersen for guiding me and giving me constructive feedbacks for my research. While being extremely busy, both have devoted much time and effort for me to create this thesis. Thank you so much.

My colleagues helped me significantly with my thesis by discussing and giving me various feedback. Especially, I thank Andrew Wesie for providing an awesome JIT engine for my research and thank Sang Kil Cha, Thanassis Avgerinos and Alexandre Rebert for the support with Mayhem. Jiyong Jang and Maverick Woo helped me immensely to develop ideas and carry out the research. I also thank Edward Schwartz for guiding me with OCaml madness! Thanks to Doug, Sangjae, and Onha for moral support. And, of course, I thank Plaid Parliament of Pwning for infinite encouragement and support.

Lastly, my parents and my brother are the ones who hold me tight whenever I have hard time. Your warm support was delivered well despite the physical distance between us. Thank you for the love and care.

Without above individuals along with numerous people I could not list, I would not have successfully done my research. Thank you, everyone!

I can do all this through him who gives me strength. -Philippians 4:13

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

There have been fair amount of work and research done in order to provide better ways to discover software vulnerabilities and exploits. The techniques that have been proposed include source code auditing, static program analysis, dynamic program analysis, and formal verification [24, 31, 10, 7, 22, 20, 39]. However, many of the techniques lie on extreme ends of the spectrum regarding the cost-effectiveness as depicted in Figure 1.1.

Static program analyses are used by many developers to test their programs because they are effective in finding some trivial bugs that can be caught by the rules that define security violations with very small resource. However, they are limited in that the performance is only good as the rules. Common technique used by many security researchers is a program testing method called *fuzzing*. Fuzzing finds bugs in a target program by natively executing it with randomly mutated or generated inputs while monitoring the execution for abnormal crashes. Fuzzing is good at quickly exploring the program code in depth because it runs the target program natively with concrete inputs. However, due to its nature, fuzzing often suffers from low code coverage problem. *symbolic execution* is another technique that has recently gotten the attention of security researchers. In contrast to fuzzing, symbolic execution tests a program by treating the program's input as symbols and interpreting the program over these symbolic inputs. In theory, symbolic execution is guaranteed to be effective in achieving high code coverage, yet this generally requires exponential resource which is not practical for many real-world programs.

Our goal is to find more bugs faster than traditional approaches. In order to accomplish this goal, we need to obtain high code coverage in reasonable resource bound (e.g. computing power and time). High code coverage implies both breadth and depth in exploration of the program. Although we may not achieve the best code coverage or speed, we aim to find the *sweet spot* in cost-effective way to gain higher code coverage than the fuzzer and higher speed than the symbolic executor as shown in Figure 1.1.

In this thesis, we present our attempt to attain the best of both worlds by combining fuzzing with symbolic execution in a novel manner. Our technique, called *hybrid fuzzing*, first uses symbolic execution to discover frontier nodes that represent unique paths in the program. After collecting as many frontier nodes as possible under a user-configurable resource constraint, it transits to fuzz the program with *preconditioned random inputs*, which are provably random inputs that respect the path predicate leading to each frontier node.

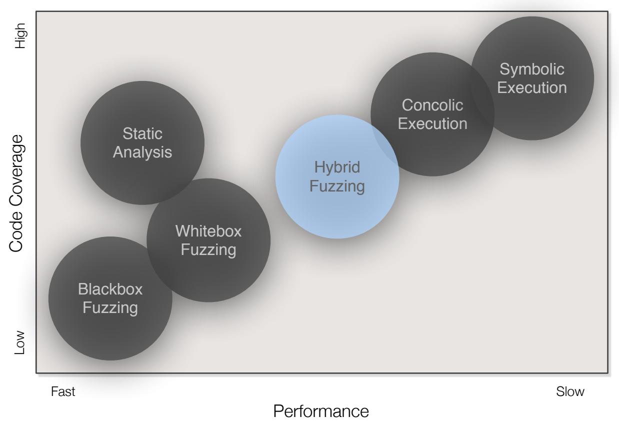

Figure 1.1: Software bug finding techniques diagram

1.2 Vulnerabilities and Bug Finding

Software bugs Software bugs are the flaws in the computer programs that are introduced by programmers. These flaws can cause various effects including the crashes, hanging or incorrect behavior of the program. The consequences of such bugs range from small inconvenience in the use of the software to catastrophic disasters where many lives and money are lost. There are several types of software bugs as there are many different programming paradigms and platforms to develop upon. While a lot of trivial bugs can be detected by static analysis tools or manual debugging stage in the development cycle, most common bugs we see these days in practice are ones categorized as *resource bugs*. Resource bugs happen when a programmer does not properly manage the creation, modification, usage, and deletion of the program resources. Null pointer dereference, uninitialized variable usage, memory access violation, and buffer overflow are common examples of the bugs of this type. Sometimes it is tricky to find such bugs because the functionality behaves as intended except for very few cases.

Security vulnerabilities Of these software bugs, we call ones that lead to denial of service (DoS) or violation of security property such as privilege escalation and arbitrary code execution *software vulnerabilities*. Vulnerability often has security implication that can be abused by malicious users who try to take advantage of program flaws. Sometimes DoS can cause a serious impact against the community such as website and/or service outage [34, 9], but it does not have the risk of possible information leakage or infection of malicious software (malware). More serious problem occurs when arbitrary code/command execution is possible. The attacker can take over the control of the victim machine, which then allow oneself to perform a secondary attack inside of the network that the victim machine belongs to. Recent cyber assaults that targeted numerous organizations also started from exploiting the vulnerability on one of the outer network machines, then moved on to the internal network for exploiting more machines and gathering sensitive information [36, 49]. In order to avoid much damage done due to the exploitation of the vulnerabilities, patching such vulnerabilities quickly is necessary. However, we need to *find* them before we can *fix* them.

There are many approaches for finding vulnerabilities. In the past, the developers and security researchers manually audited the source code (if available) or the disassembly of the binary programs. However, this method does not scale as the program gets complicated. Advance in automated systems that check for vulnerabilities in the program quickly opened the path of efficiently finding bugs. Some of the automated methods actively being used by the developers and security researchers are discussed and explained in Chapter 2.

1.3 Thesis Outline

This thesis consists of several chapters. While this chapter covers general terms and insights on existing bug finding techniques, we describe in more detail bug finding approaches in the second chapter. Specifically, we explain commonly used analysis techniques in both static and dynamic analysis realm.

Starting from Chapter 3, we delve into the new technique we propose in this thesis, called *Hybrid Fuzz Testing*. We describe the intuition and overall architecture and design of the new hybrid fuzzing system. Then, we move into specific problems that we encountered while developing hybrid fuzzing and discuss our solutions. We describe the core concept of $\mathbb{Z}$-Polytope abstraction for efficient input generation in Chapter 4.

Evaluation is followed to present the effectiveness of the hybrid fuzzing technique. In this chapter, we provide the statistics on formula transformation and input generation. Then, we compare the code coverage with other known techniques for discovering vulnerabilities followed by the experimental results on three synthetic programs and twenty real-world x86 Linux utilities (including 15 coreutils). Finally, we present observations on these results to explain the effectiveness of our method.

Remaining chapters include discussion of our work, limitation, and future work along with the conclusion we learned from the research in developing hybrid fuzzing system with efficient input generation mechanism.

Chapter 2

Software Bug Finding Approaches

There have been much research done on developing and improving the methods to make the programs more secure by finding the mistakes that are introduced by programmers automatically [25, 7, 41, 19, 15, 1]. However, finding *all* bugs in a reasonably sized program is infeasible. With this realization, researchers from both academia and industry have come up with several techniques that can be applied in order to quickly locate as many software bugs as possible. In this chapter, we present commonly used *automated* approaches for finding bugs and discuss their strength and limitations.

2.1 Static Analysis

In this section we explore two different types of static program analysis techniques that have been introduced and researched. Some techniques presented here are used in practice in order to find critical security bugs in the programs. We illustrate the details and the application for each analysis with couple examples.

2.1.1 Static Code Analysis

Static code analysis is a method to detect potential coding errors in the program source code without actually executing the program. Usually the analysis is done by the automated tools that implement rules that are checked against the code to verify if any portion violates them. Depending on the modeling, static code analysis can find from trivial bugs (but easy to miss) to complex one, which requires understanding of cross file/class relationship. Static code analysis is powerful because it is fast and cheap while generally effective. It can provide detection of flaws in the software that dynamic analysis cannot easily expose.

Static code analysis is usually implemented by parsing the source code and building an abstract syntax tree (AST) of the program. Since modern compilers already perform the above job, sophisticated tools extend upon this. For example, type checking and data-flow analysis are performed. Common use case of static code analysis is in the realm of discovering buffer overruns, format string vulnerabilities, and integer overflows as suggested in [45].

```
1 char *data = (char *)malloc(16); // data can be NULL
2 memcpy(data, input, 16);
```

Listing 2.1: Possible NULL dereference bug

Consider the code shown in 2.1. This piece of code contains a possible bug for NULL pointer dereference because `malloc` call on line 2 may return NULL, in case of failure. Static code analysis can spot these errors and suggest programmers to insert a NULL checking routine as shown in 2.2.

```
1 char *data = (char *)malloc(16); // data can be NULL
2 if (data == NULL) abort();
3 memcpy(data, input, 16);
```

Listing 2.2: NULL pointer check

Static code analysis is very effective when finding incorrect usage of programming idioms such as format string bugs. In the code 2.3, on line 2, the user controllable data is directly passed to `printf` function as the first argument, which represents the format string. This works without a problem when the passed string does not contain any format specifier. However, when the user (or attacker) can control the contents of the string arbitrarily, serious security flaw is introduced that can be abused [32, 38]. Static code analysis can discover this type of error easily and suggest a fix as shown on line 3. With the type checking feature, it can also provide the appropriate format specifiers based on the passed arguments.

```
1 char *buf = get_user_input();
2 printf(buf); // Potential format string bug
3 printf("%s", buf); // Correct usage of printf
```

Listing 2.3: Format string bug

Another example of finding bugs with static code analysis is discovering integer overflow vulnerabilities. Integer overflow occurs when signed operation is used when unsigned operation is intended or arithmetic operation results in an overflow. On line 3 in 2.4, `nresp` multiplied by the size of `char *` can have integer overflow and becomes 0 with carefully chosen `nresp` value. Then, the following loop can overflow the heap since the size of allocated `response` buffer is actually 0. It is usually tricky to find this type of bugs manually because there are only few cases where the bug is triggered. However, it is indeed a type of bug that can lead to serious security problem [4].

```
1 nresp = packet_get_int();
2 if (nresp > 0) {
3    response = xmalloc(nresp*sizeof(char*)); // Potential Int Overflow
4    for (i = 0; i < nresp; i++)
5       response[i] = packet_get_string(NULL); // Potential Heap Overflow
6 }
```

Listing 2.4: Excerpt from OpenSSH 3.3

However, the limitation of static code analysis soon becomes obvious. It usually does not support all programming languages, and introduce many false negatives and false positives. Also, the static code analysis is only as helpful as the rules that are employed. More importantly, static code analysis cannot reason about the concrete values for which are generated dynamically. Therefore, when the analysis does not find any bug, it does not imply that the target program is bug-free.

2.1.2 Static Binary Analysis

Static binary analysis is based on the same concept as static code analysis where it does not involve concrete execution of the program, and checks any predefined rule is violated in *binary code* as opposed to source code. Thus, it shares a lot of advantages and limitations of static code analysis. However, since the platform is totally different (source vs. binary), the engineering mechanics change in some degree.

Static binary analysis is quite different from static code analysis in that the context we need to deal with is not in high-order languages, but rather compiled to low-level assembly language for architecture specific programs or bytecode for platform-independent programs such as Java or .NET. Also, unlike static code analysis, extracting and constructing useful information from the program such as control flow graph may be challenging especially when the symbols are missing or in case of indirect jump instructions. On the other hand, static binary analysis can reveal errors that are introduced by compilers as a part of optimization process, which is important because this type of information is hidden or not available in static code analysis. However, the underlying approach to discover flaws remains the same where the code is transformed into some intermediate language and faulty chain of operations is searched.

There are few researches in academia that advanced this field [44, 6]. Usually the tools are targeted for x86 architecture, but the analyses are platform-independent since they are performed in the level of intermediate language (IL). There are many non-trivial design decisions to make because static binary analysis tools need to understand all possible opcodes that are in a certain instruction set architecture (ISA) and realize the correct semantics of each instruction.

In some cases, static binary analysis platform is used to implement taint analysis on top. Taint analysis needs to keep track of what registers and memory regions are tainted or clean throughout the program since the initial user-controllable input is introduced. However, due to the challenges such as indirect jump instructions and memory operations (since it is static analysis, there is no dynamic, concrete values available), it is often not practical to perform such analysis with real programs and requires manual investigation (by human) [43].

Some researches target specifically for finding the software vulnerabilities in the programs [12, 13]. Even with these advanced techniques, however, static binary analysis may not be sufficient to understand the program and discover bugs when there are compiler optimizations or code obfuscation that the analyzer does not handle [35].

2.2 Dynamic Analysis

In order to overcome the limitations of static analysis, another program analysis paradigm is suggested: dynamic analysis. In this section, we describe a couple major approaches in dynamic program analysis and discuss their strength and limitation. Dynamic program analysis technique, due to its nature, tends to have a significantly higher overhead compared to static analysis technique. With the cost of performance and resource, however, dynamic analysis provides extremely useful information such as concrete values and dynamic address resolution.

2.2.1 Fuzz Testing

Fuzz testing, or *fuzzing* for short, is a dynamic program testing method that effectively finds software vulnerabilities by feeding malformed or unexpected data as input to the programs. It aims to discover cases that programmers have missed in testing such as extreme or nonsensical values. Fuzzing achieves its goal by feeding generated or mutated inputs to the program and monitors the effects on the target programs, especially for crashes.

There are many types of fuzzing based on the input generation mechanisms. Generally, we categorize the fuzzing techniques as follows:

- Input Generation
  - **Mutation-based:** Either deterministically or probabilistically modify parts of the input file or arguments as a means of generating test cases. This requires known valid *seed* files, and especially good seed files that will force the program to exercise various code. Many available fuzzers employ this technique by default since it is most intuitive and simple way of generating large test cases quickly [37, 27, 14]. Listing 2.5 shows an example of mutated GIF file that is derived from the original GIF file.
  - **Generation-based:** Instead of mutating the available seed files, the files are automatically generated based on the *template (grammar)* that describes the file's syntax. This type of input generation requires a good modeling of the input, which requires thorough understanding of the specification to be effective [30, 14].
- Context Knowledge
  - **Blackbox fuzzing:** The fuzzer does not realize the internals of the target program in this type of fuzzing. Generated (known) files are fed into the program and the fuzzer monitors the output (behavior) of the program. It is mostly taken by default approach in many available fuzzers due to simplicity and performance. However, blackbox fuzzing suffers from code coverage problem since it does not reason about the process of the execution – thus does not realize that it's looking at the same portion of the code all the time.
  - **Whitebox fuzzing:** The fuzzer takes an advantage of the target program's internal structures or design in this type of fuzzing. Fuzzer usually keeps track of the state and each fuzz run has its goal to test the correctness of a certain portion of the code iteratively. The goal may be maximizing the code coverage or testing extreme values for a certain function. Whitebox fuzzing usually gets assisted by other techniques such as symbolic execution or taint analysis to decide on what to do next [22, 16].

```
00000000  47 49 46 38 39 61 10 00  10 04 aa 84 01 00 02 00  |GIF89a..........|
00000010  df fe ff 84 89 01 c0 c0  c0 ff ff ff 04 00 00 08  |................|
00000020  10 00 10 00 00 28 b9 06  81 10 1c 06 00 a8 02 00  |.....(..........|
00000030  41 40 09 00 10 00 80 83  38 4a 0b d8 1e 2a a2 40  |A@......8J...*.@|
00000040  6b 0d 63 82 69 6d 93 5d  f8 8c c4 74 9c 20 ca 08  |k.c.im.]...t. ..|
00000050  15 1b 9c 94 bc b3 f6 23  6b 49 7d 8f 34 7f f3 7d  |.......#kI}.4..}|
00000060  2a 6d 15 13 ee 66 b9 94  32 49 6a 82 1a 9f 27 14  |*m...f..2Ij...'.|
00000070  00 59 b4 79 09 00 13                              |.Y.y...|

00000000  47 49 46 38 39 61 10 00  10 00 a2 04 00 00 00 00  |GIF89a..........|
00000010  ff ff ff 80 80 00 c0 c0  c0 ff ff ff 00 00 00 00  |................|
00000020  10 00 10 00 00 21 f9 04  01 00 00 04 02 2c 00 00  |.....!.......,..|
00000030  00 00 10 00 10 00 80 83  3c 48 0a dc 0e 2a a2 40  |........<H...*.@|
00000040  6b 05 23 92 69 6d 93 5d  f8 8c a4 60 9e 28 ca 08  |k.#.im.]...`.(..|
00000050  15 1b 9c 94 b9 b2 e6 6b  ab 40 5d bf 34 3f f3 3c  |.......k.@].4?.<|
00000060  1a 6d 15 13 ee 64 b9 94  32 49 6a 82 42 9e 27 14  |.m...d..2Ij.B.'.|
00000070  00 59 34 1d 09 00 3b                              |.Y4...;|
```

Listing 2.5: Original GIF vs. Mutated GIF

Fuzzing has been discussed extensively in both academia and industry and described as one of the most effective approach of finding bugs [48, 46, 42]. There also have been numerous heuristics suggested in trying of maximizing the effectiveness of the fuzzing and eliminating some of the limitation that fuzzing suffers from [50, 20]. Some researchers compare and contrast on effectiveness of various fuzzing approaches [33, 18].

2.2.2 Symbolic Execution

Symbolic execution is originally used as a means of program verification to *prove* the program's correctness. Researchers recently have started shifting the focus of the symbolic execution from the formal verification to vulnerability analysis of the program. Symbolic execution works by supplying and tracking abstract *symbols* rather than concrete values in the hope of generating the symbolic formulas over the input symbols [26].

Pure Symbolic Execution There are a few different approaches in performing symbolic execution. Based on the mechanics, they can be categorized as offline, online, and hybrid symbolic execution. Details on each approach are as followed.

- **Offline Symbolic Execution:** A type of symbolic execution where the input is generated by solving path predicates that the executor has faced in the symbolic execution phase. Then, in order to find new input (or path), the executor is again run from the beginning of the program and flips the predicate to force the execution to the different path from previous run. However, this usually entails execution overhead due to re-executing the program symbolically from the beginning for finding new paths. Good application of offline symbolic execution is SAGE, developed by Microsoft Research [22].

- **Online Symbolic Execution:** A type of symbolic execution where the states are duplicated and the predicate is flipped for each branch the executor encounter. By doing so, online symbolic execution does not require multiple runs of the program from the beginning. However, because most of the state information need to be stored and accessible in online fashion, the memory usage becomes immediate problem, especially for large programs. An application of online symbolic execution is KLEE from Stanford University [8].

- **Hybrid Symbolic Execution:** In order to overcome the limitations of offline and online symbolic execution techniques, a new model is proposed, in which we combine the both worlds. In this hybrid scheme, the *seed* file is produced when the branching point is met. This seed file ensures the execution till that point of the code when fed to the program. This way, it achieves better memory usage compared to online symbolic execution due to state saving to disk as a prefix file, and also runs performs path exploration faster than offline it does not need to run the program symbolically for the path that the stored prefix (seed) file restores. This is implemented in symbolic execution engine for Mayhem platform from Carnegie Mellon University [40].

Concolic Execution Unlike pure symbolic execution, concolic execution is a hybrid software verification technique that combine concrete execution and traditional symbolic execution. That is, the execution of the program is initiated with some concrete input (but also marked as symbolic) and follow the paths that chosen input lead to. When the execution encounters conditional branches, the *path predicates* are constructed to be used to generate other concrete inputs that lead to different paths by flipping the condition on each symbolic variable [19, 41, 7].

Here, we demonstrate a simple example of symbolic execution and show how it can be used to reveal vulnerabilities and possible exploits. Consider the following program shown in Listing 2.6 that reads an integer from a user and prints out the value of x and y if the `Path 2` and `Path 3` are reached, respectively, but the bug is triggered when the `Path 1` is reached.

```
int main(int argc, char *argv)
{
   int x = user_input(); // User-controllable data introduced
   int y = 3*x;

   if (y <= 15) {
      if (x >= 0) {
         // Path 1
         trigger_bug();
10            return 0;
11        }
12        // Path 2
13        printf("%d\n", x);
14        return 0;
15    }
16    // Path 3
17    printf("%d\n", y);
18    return 0;
19 }
```

Listing 2.6: Simple program

Usually the variables that are associated with user-controllable input (such as files, arguments, and data from socket) are marked as *symbolic* and be called *symbolic variables*. In the above program, x initially becomes a symbolic variable since the value is determined by the user input on line 3. As explained earlier, these symbolic variables and expressions that involve them are tracked by the symbolic executor. Now, on line 4, new variable y appears and is marked as symbolic since y is derived from the expression $3 * x$, which involves the symbolic variable x. After this line, symbolic variable y contains the expression $3 * x$, rather than the concrete value. On line 6, the executor faces the branch and builds a *path constraint* for both branches – in this case, $y \leq 15$ and $\neg(y \leq 15)$ – and update the constraint context. Then, the executor moves on to the next line of the code to find another branch. Similarly, following path constraints are built: $\{(y \leq 15) \wedge (x \geq 0)\}, \{(y \leq 15) \wedge \neg(x \geq 0)\}$. There are no more branches left in the program, thus symbolic executor explored all possible paths. The final constraint context contains three path constraints, $\{(y \leq 15) \wedge (x \geq 0)\}, \{(y \leq 15) \wedge \neg(x \geq 0)\}$ and $\{\neg(y \leq 15)\}$, which correspond to `Path 1`, `Path 2`, and `Path 3`, respectively.

In order to generate concrete inputs that will explore these three unique paths, the theorem prover is queried with each of the path constraints and asked for the solution, if solvable. Depending on the algorithm of the solver and complexity of the constraints, the solver may not respond in a reasonable time. For the formulas we have in this example, the satisfying solution can be computed quickly. A solution is an assignment of the values for all symbolic variables in the context such that it satisfies the given formula (constraint). Note that there are more than one solution for each path in the example program. The solvers will return a set of assignment, thus generating one valid concrete input that will lead to a certain path that the given path constraint represents. One possible set of solutions that will reach all three paths is 2 for `Path 1`, -1 for `Path 2`, and 35 for `Path 3`.

Chapter 3

Hybrid Fuzz Testing

3.1 Overview

Our intuition on hybrid fuzzing comes from the obvious differences in characteristics of symbolic execution and fuzzing. Symbolic execution is capable of discovering and exploring all possible paths in the program, but is not practically scalable since the number of the paths quickly becomes exponential. While fuzzing is much faster than symbolic execution and thus guarantee to explore deeper portion of the code, it has limited code coverage in breadth. Figure 3.1 depicts this difference by showing the explored code for each method in given time.

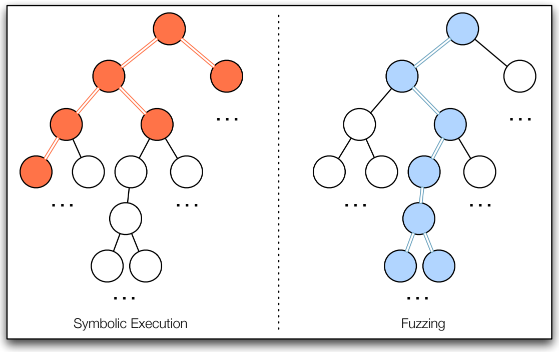

Figure 3.1: Code exploration comparison between symbolic execution and fuzzing.

The goal is to take advantage of symbolic execution to spread out to various different (unique) paths, then use fuzzing to quickly test each path. Of course, we may not be able to cover entire program since the symbolic execution is stopped after exploring only part of all paths in the program, then the system would transit to fuzzing, which does not necessarily guarantee the exploration of all possible paths. By discovering as many different paths as possible soon after the user data (e.g. command-line arguments, files, data from network sockets, etc.) is introduced, and by ensuring the generated random inputs follow each path, we can have similar effect of performing the fuzz testing on many unique paths without having to collect good set of seed files to begin with.

Hybrid fuzzing consists of four phases: `Basic block profiling`, `Symbolic execution`, `Input generation` and `Guided random fuzzing`. In the first two phases, we collect path predicates for the target program, extract linear predicates to build polytopes that describe the input space for each path. Then, we efficiently generate random test cases that respect path predicate. In the last phase, we feed the generated inputs to the target program and monitor its behavior. Also note that last four phases can be pipe-lined for better performance.

Basic Block Profiling *Basic block* is a sequence of instructions with exactly one entrance and one exit. This means that we can safely assume that if the first instruction in a basic block is executed, the rest of the code in the same basic block is executed as well. Also, basic block is closely related to the code coverage measurement because the executed basic blocks essentially represent the portion of the code that are covered in the program by the given input.

We often run into the situation where the executions are *stuck* to limited code paths and seed files do not span out for higher code coverage. Traditional way of exploiting this problem is to increase the number of *better* seed files that will exercise more unique program paths. However, it is usually not practical due to the limited resources/samples available. The intuition behind `Basic block profiling` phase is that the portion of the code that are visited with initial (possibly limited) seed files will likely be explored easily without help of more expensive techniques such as symbolic execution. Thus, we collect the union of basic blocks that have been executed during these executions to allocate more computing resources to reveal the code paths that have not been executed in the next phase, `Symbolic execution`.

In case the source code of the target program is available, we can recompile the program such that we can easily track the basic blocks that are executed using such tool as *gcov* [17]. However, in order to also handle the binary programs, we use *dynamic binary instrumentation (DBI)* to profile which basic blocks are executed by the program with given inputs. We adopt a custom JIT (Just-in-time compilation) engine to dynamically instrument the target programs for profiling executed basic blocks. This gives us two benefits: capability to handle binary only targets and competitive performance over publicly available DBI tools. More detail is explained in Section 3.2.

Symbolic Execution In this phase, we perform an online symbolic execution of the program with the inputs as symbolic variables. The goal is to explore as many paths as possible within the specified resource constraint to increase the breadth of covered code in the program. We are using a binary symbolic executor in order to support not only programs with the source code, but also compiled binaries. The symbolic executor is designed such that the basic block profile from the previous phase can be imported to be used for path selection. Specifically, the paths that are not included in the profile are assigned higher priority so we can reveal more paths that are not explored before.

For flexibility of the system, some of the parameters can be configured. One of the parameters that is critical is the scope of the path exploration. When a developer or a security research is testing an application for any security vulnerabilities in the program, the target is usually focused to the application itself. Thus, we provide a way to configure to avoid other code base such as shared library code as much as possible during symbolic execution. Another important parameter is frontier node threshold. Frontier node represents the basic block that a certain executor is trying to execute next. Thus, if the threshold for the number of frontier nodes is set to the number of all possible paths in the program, it would essentially be the same as full symbolic execution of the program. Since we are aiming to attain *some* breadth, the threshold does not need to be set too high. It is basically the number of unique paths to generate test cases for.

For each path the executor takes, it dynamically builds the path predicate. Therefore, each path predicate represents a unique path in the program, and the solution to the formula is an input that leads the program execution to that certain path. So, the path predicates built up to frontier nodes describe the conditions to be satisfied in order to get to the end of each path (i.e. frontier node).

Input Generation In order to perform fuzz testing, we need to be able to automatically generate random inputs that will respect the path predicates to each frontier node. The key idea is to introduce as much as randomness to the generated inputs such that we obtain similar effects as running random fuzzing, which is known to be quite effective, on execution beyond the frontier nodes. We proceed to generate such inputs by transforming the input possible input space for each path into $\mathbb{Z}$-Polytope representation.

We first transform the linear formulas we obtained from the `Symbolic execution` phase into the system of linear inequalities, called *constraint system*, using the transformation rules we have defined. We mark the symbolic bytes that are referenced by a given formula as *path critical* bytes, meaning that these bytes are constrained and must satisfy the condition. The remaining non-path-critical symbolic bytes can be any value and still would not affect the possibility of getting down to the same path.

To represent precise input space for a given formula, we build a polytope described by the constraint system transformed from the formula. Polytope is a geometric object with bounding facets. In this thesis, we will be using the term polytope for *convex polytope* with bounded convex set of points. n-Polytope is an n-dimensional polytope. For instance, a polygon is 2-polytope, a polyhedron is 3-polytope, and so on. Also, in our context, a polytope is considered to be *convex lattice polytope ($\mathbb{Z}$-Polytope)* that is a convex hull of finitely many integer points. Finite set of linear inequalities, all of which are satisfied by the concrete values, can be used to build the convex polytope [11].

Not surprisingly, there is no efficient algorithm known for enumerating all possible integer points (which are the solutions to the formula in our case) for a given polytope. Thus, we build a hyperrectangle that inscribes the polytope. Hyperrectangle is a generalization of a rectangle for higher dimensions. Thus, n-hyperrectangle is basically an n-polytope with all edges being mutually perpendicular, or an *orthotope*. Although *hypercube* is a specific type of hyperrectangle with edges of equal length, we may refer them interchangeably. Finding such hyperrectangle is trivial given a polytope, and so is to choose random points in the hyperrectangle. Note that, however, the point chosen from the hyperrectangle is not necessarily contained in the inscribed polytope. In order to randomly select a point that resides in the polytope, we perform a rejection sampling, where we test if a chosen point indeed satisfies the constraint system that describes the polytope. If we find a point that passes the test, that point is a valid input for the path. Also, number of dimensions needed to represent the polytope (and thus, hyperrectangle) depends on the number of *path critical* bytes.

Guided Random Fuzzing The remaining parts of the hybrid fuzzing system consist of executing the target program with the generated random test cases. The fuzzer acts as a concrete executor that simply feeds the input to the program and monitors its behavior. When abnormal state such as crashes and hanging is detected, the fuzzer reports with the faulty input that triggered the bug. The state to be monitored can be configured such that only *segmentation faults* are reported, for example. Note that each input is randomized as much as possible while still being able to reach the frontier node that the input is generated for.

3.2 Architecture and Design

Basic Block Profiler For profiling the executed basic blocks over the initial seed files, the hybrid fuzzing system uses a light-weight custom JIT engine for 32-bit Linux binary programs, written by Andrew Wesie. It is trivial to extend to other architectures by using more advanced DBI tools such as PIN or DynamoRIO [31, 5], but the choice was made due to huge performance improvement. Our custom profiler consists of total 1500 lines of C and x86 ASM. The profiler injects itself to the target program's memory and monitors every control transfer during the target program's execution. Then, it emits arbitrary code that we want to execute before the program continues with its own code.

Basic block in our tool is defined the same as in PIN instrumentation, where a basic block is a single entrance, single exit sequence of instructions. The profiler discovers the control flow of the program as it executes as PIN does. For example in 3.1, we find either `BBL 2` or `BBL 3` based on the value of the first argument to the program, but never at the same time.

```
80483b4:        55                      push   %ebp                ; BBL 1 Start
80483b5:        89 e5                   mov    %esp,%ebp
80483b7:        83 ec 10                sub    $0x10,%esp
80483ba:        8b 45 0c                mov    0xc(%ebp),%eax
80483bd:        8b 40 04                mov    0x4(%eax),%eax
80483c0:        89 45 fc                mov    %eax,-0x4(%ebp)
80483c3:        8b 45 fc                mov    -0x4(%ebp),%eax
80483c6:        0f b6 00                movzbl (%eax),%eax
80483c9:        3c 63                   cmp    $0x63,%al
80483cb:        76 10                   jbe    80483dd             ; BBL 1 End
80483cd:        8b 45 fc                mov    -0x4(%ebp),%eax     ; BBL 2 Start
80483d0:        0f b6 00                movzbl (%eax),%eax
80483d3:        8d 50 0a                lea    0xa(%eax),%edx
80483d6:        8b 45 fc                mov    -0x4(%ebp),%eax
80483d9:        88 10                   mov    %dl,(%eax)
80483db:        eb 0f                   jmp    80483ec             ; BBL 2 End
80483dd:        8b 45 fc                mov    -0x4(%ebp),%eax     ; BBL 3 Start
80483e0:        0f b6 00                movzbl (%eax),%eax
80483e3:        8d 50 ec                lea    -0x14(%eax),%edx
80483e6:        8b 45 fc                mov    -0x4(%ebp),%eax
80483e9:        88 10                   mov    %dl,(%eax)
80483eb:        90                      nop
80483ec:        c9                      leave
80483ed:        c3                      ret                        ; BBL 3 End
```

Listing 3.1: Basic Block example

For our purpose, we simply logged the starting addresses of the basic blocks and the counter which keeps track of how many times that basic block is executed. The BBL data is written to a file that is later imported by the symbolic executor. Using the above example program, we can obtain a table as shown in Table 3.1. For this profiling, we have executed total of 5 seed inputs where three of which reached `BBL 2` and the other two reached `BBL 3`. Obviously, `BBL 1` is reached total of five times during the executions since this basic block is executed regardless of the input.

BBL Address	Count	# instructions
80483b4	5	10
80483cd	3	6
80483dd	2	8

Table 3.1: Basic block profiling of the program shown in Listing 3.1

As a small optimization for both performance and space, we log the profile information in binary format rather than in string format. Also, because our tool is minimally set up to achieve the goal, it does not add any unnecessary overhead as other feature-full DBI tools do such as operand extraction. Table 3.2 shows the runtime comparison between our tool and Pin tool, where both tools perform the same job: basic block profiling.

Application	Native (sec)	PIN (sec)	Custom Engine (sec)
gzip (decompress 432MB)	38	353	154
gzip (compress 1.7GB)	101	2199	855
md5sum (1600 files)	0.73	22.42	1.79
exim-4.41 (100 runs)	0.44	193	2.21

Table 3.2: Runtime comparison between PIN and our custom JIT engine

Usually, the cost of running complicated programs under Pin tool is too high that one can argue running a mutational blackbox fuzzer would be a better choice for the same time it takes for Pin tool to profile over the seed files. However, with our custom DBI tool, we can efficiently profile executed basic blocks to aid the symbolic executor for discovering more useful (i.e. not likely hit by concrete testing) paths. Also, we log all executed basic blocks including the shared library code. The decision to either include or exclude library code section to the scope of path exploration can be made in `Symbolic execution` phase.

Symbolic Executor Symbolic executor of our choice for the hybrid fuzzer is Mayhem symbolic executor. We use it to perform a pure online symbolic execution where it does not generate any test case, thus not invoking the theorem prover at all. Because we do not explore all paths in hybrid fuzzing system, pure online symbolic execution does not yield a problem of memory exhaustion.

Path selection in Mayhem is based on the ranking of each available executor in the queue. Ranking is determined by the *score*, which is increased or decreased depending on the objective. For instance, if we were to maximize the number of explored paths, then the

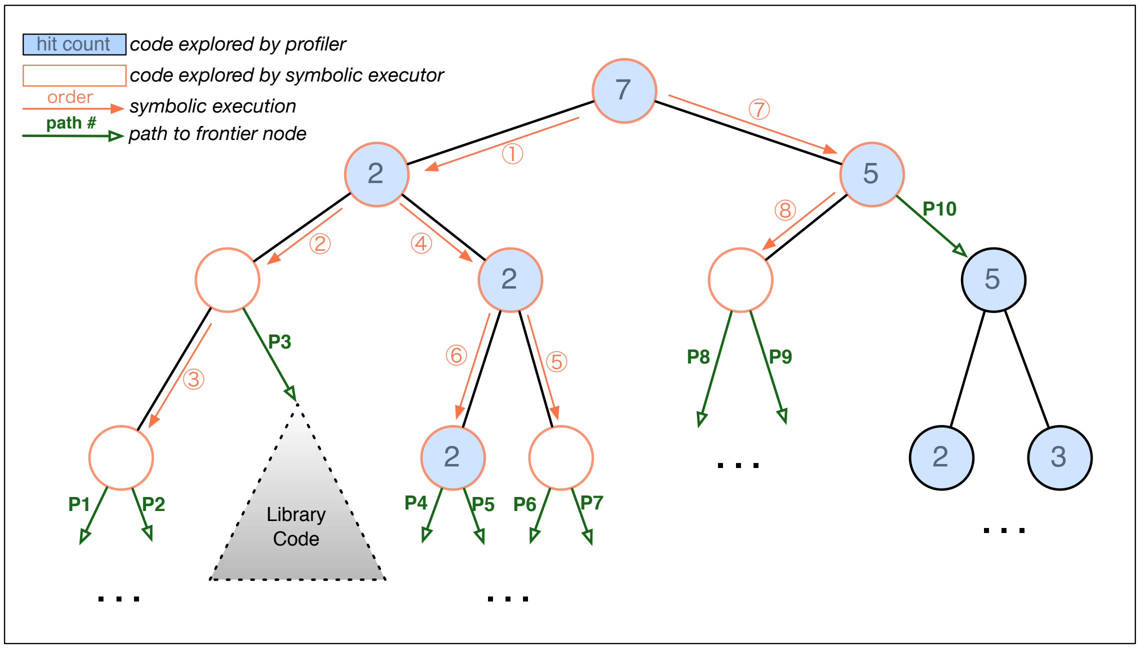

Figure 3.2: Path selection based on BBL profile and target addresses.

score of the executor is increased when a certain executor has found a path that we have never explored before. Thus, the symbolic executor manages a priority queue that will dispatch the next executor based on their scores. Also, in order to prevent one executor exploring too deep alone, the maximum depth difference is maintained. If the scope of the path exploration is limited to the main executable itself, we assign higher score to the paths that reside in the program text and lower score to any other library codes such as *libc*. This, however, can easily be relaxed in case the user wants to allow the symbolic executor to discover paths in the libraries that are loaded with the main executable.

Modified Mayhem first imports the basic block profile that the profiler generated and looks up whenever the executor needs to fork another executor (e.g. the conditional branch is met). Unless the threshold for the number of frontier nodes has been reached, Mayhem updates the score for the executors based on two factors: current depth and basic block hit count. Figure 3.2 shows the process of path selection in Mayhem with imported profile information. In the example above, the maximum depth difference and the maximum number of frontier nodes are set to 2 and 10, respectively. Also, the path selection configuration tries to avoid any code other than the main executable.

Blue-shaded nodes in the figure represents basic blocks that are explored by the profiler and thus Mayhem has hit counts for, which is shown by the number on each node. When the symbolic execution reaches the root node in this diagram (note that this tree may be a sub-tree of larger execution tree of the program), the executor needs to fork and updates the score for each path. The profile shows left branch is less visited by the profiler (2 out of 7 times), so Mayhem decides to explore left path first as the orange arrow with the order 1 suggests.

After a while, the symbolic executor forks again and sets higher priority to left branch (marked by symbolic execution of order 2) because the left path is never explored by the profiler whereas the right path is explored (executed) twice by the profiler. There is another branching point afterward, but the symbolic executor prefers the left branch (symbolic execution of order 3) since the right branch takes the execution to the library code which is outside of the main executable code. Then, Mayhem sees the largest difference between the current executor and delayed executor (in the queue) has reached the threshold – 2, in this case –, so it stops exploring further with the current executor. Instead, Mayhem resumes the execution of the best (i.e. highest score) executor from the queue, which is execution to the path that is marked by symbolic execution of order 4 in the example above. The rest of the paths are explored in the same manner until Mayhem reaches the frontier node threshold, which is set to 10 as shown as green paths with the label (P1−P10).

As a result, this process generates total of 10 paths while preferring the discovery of the code that is not explored by the profiler and avoiding the code segments that are not of our interest – when all of the remaining (delayed) executors are targeting non-main executable code paths and the frontier node threshold is not met, Mayhem will still explore those paths.

When an executor terminates, the path constraints built up to that point of the program are combined into one formula that essentially represents the path that the executor explored. There are two conditions under which the executors terminate. The first is when the executor reached the end of the path (i.e. the leaf note in the execution tree), and the second is when pre-defined number of frontier nodes (i.e. unique paths) is reached. Every time an executor terminates and a path formula is constructed, the formula is sent to *Input Generator* to be parsed, transformed and represented as a $\mathbb{Z}$-Polytope defining the input space for the given formula.

Because inputs are generated as the symbolic execution is still running, tasks can be pipe-lined for the *Guided Fuzzer* to receive generated test cases from input generator, feed them to the target program, and observe for anomaly. This is important since the bottleneck of the entire hybrid fuzzing system is the symbolic execution, which we limit its resource of.

Input Generator As a numerical abstraction of input space, we use linear inequalities and the corresponding convex polytope generated by the inequalities. The input generation module is developed upon Parma Polyhedra Library (PPL), which is developed and maintained by University of Parma, Italy [3]. It generates pre-configured number of test cases for each time it is invoked with a certain formula. Input generation module consists of 940 lines of OCaml code that uses PPL's OCaml interface.

Input generator first extracts linear predicates from the passed formula. Then, we use PPL to build the constraint system by transforming the linear predicates to non-strict linear inequalities that represent a *closed* polytope. Note that in our setting, any generated polytope will be bounded because we know the maximum and the minimum values possible for each variable. For instance, the implicit lower bound and upper bound would be 0 and 255 for the variable mapped to an unsigned byte. Most importantly, the library provides an efficient way of performing rejection sampling. We can build a linear generator expression of a point in n-dimensional space, where n is the number of symbolic variables. Then, this point can be quickly tested if it is contained in a certain polytope (e.g. polytope, powerset of polytopes, boxes, etc.).

We currently support simple formulas that involve only up to 2 symbolic variables in a single expression with the symbolic variable being signed or unsigned 8-bit and 32-bit integers. However, extending the system to handle more complex expressions is as easy as adding more rules for transformation as we will show in Chapter 4.

Only path critical symbolic variable are mapped to a PPL variable so we can reduce the required dimensions to produce a polytope of input space. For other symbolic variables that are not path critical, the input generator assign random values to them. Since each variable with lower and upper bound in the polytope data structure consumes about 360 bytes, the less there are variables for PPL to track, the less memory is used by the input generator. However, this still gives quite flexible ranges of input size such as 300 KB files which will require little above 100 MB of RAM. Note that the resource allocated for polytope is freed when the sampling is finished.

Next step in `Input Generation` phase is to perform a rejection sampling. We define a generator for a point, where the coefficients are determined randomly in the valid range for each variable. Then, the provided method from PPL is used to check the relation of the polytope with a generator. If the returned type is *Subsumes*, then we can conclude that the chosen point is contained inside of the polytope – and thus the valid input. We repeat random rejection sampling until we reach the pre-configured number of test cases to be generated per frontier node. The generated inputs are stored to disk such that the *Guided fuzzer* can later feed to the target program and monitor.

Guided Fuzzer / Monitor The fuzzer does not actually mutate the inputs, but rather simply feed to the target program generated inputs that are outputted by the input generator. Also, this fuzzer needs to behave as a monitor that looks for crashes and hangs. For this purpose, we use *Zzuf* fuzzer with mutation ratio of 0% [27]. Then, the hybrid fuzzing system organizes the crashes into separate directories along with the crashing inputs for further analysis. Since most of the *interesting* bugs cause the crash of the program, we configure the fuzzer to monitor for *segmentation faults*. We do not perform any optimization to remove duplicate crashes, but this can be easily done by adding some sort of hash for the crashes based on the crashing location, stack trace, etc.

Chapter 4

Efficient Input Generation

The process of generating provably random inputs that are likely increase the code coverage while preserving the path predicates for frontier nodes is critical. However, efficiently generating such inputs is not a trivial task. In this chapter, we present the abstraction of the possible input space and provide an algorithm to efficiently produce random inputs that respect the falsifiability of the given path predicates. This corresponds to `Input Generation` phase where we use Parma Polyhedra Library as we stated in previous chapter.

4.1 Path Predicates

We call an input *path-preserving input* if it satisfies the path predicate so the execution to that path is guaranteed when running a such input. Our purpose is to produce as many path-preserving inputs as the user specified without invoking any formula solvers. In fact, we want to introduce as much randomness as we can while building such inputs for *fuzzing effect*, which has shown its effectiveness [47].

We first consider the *linearity* of the formulas, since our abstraction is built with an assumption of linear expression. By linear expression, we mean the expression that only involves either constants or first-order variables with constant coefficients. In the context of formulas, a linear formula is a formula that only involves constants and first-order symbolic variables. For example, `symb_0 <= 3·symb_1 + 0x70` is a linear formula whereas `symb_0·symb_1 >= 0x32` is not. Fortunately in many programs, path predicates are fairly simple and linear. This is a sample formula generated with one of the coreutils, *id*:

```
  symb-argv_1_0:u8 == 0x2d:u8 & ~(symb-argv_1_0:u8 == 0:u8)
& ~(symb-argv_1_0:u8 == 0x2b:u8) & symb-argv_1_0:u8 == 0x2d:u8
& ~(0:u8 == symb-argv_1_0:u8)
```

Listing 4.1: A sample formula from *id*

Predicates usually encode the ranges of values that each input byte can have. If we can precisely describe these ranges and efficiently pick values in valid range, we can generate path-preserving inputs. For instance, `(symb_0 == 0x62 & symb_1 < 0x10)` represents the condition where the first symbolic byte has to be equal to value 0x62 and the second symbolic byte must be less than the value 0x10 at the same time. While solvers can return a solution such as `[symb_0 <- 0x62; symb_1 <- 0x9]`, there are 137 more values that `symb_1` could have been with signed symbolic bytes. Therefore, we gain significant benefit in generating path-preserving inputs by extracting linear relationships (ranges) from the formulas as opposed to querying solver repeatedly to obtain a solution at a time.

Also, we perform several formula simplifications prior to process the formula for linear expression transformation. We currently perform simplifications for *and_with_true, xor_with_self, less_than_eq, equal_to_val, plus_zero, pad_zero, is_zero*. Most of the original formulas ended up more complicated than they need to be due to the way intermediate language is generated based on the disassembly of the program. The simplification code is shown in Appendix A.

There exist some non-linear formulas that can be transformed to linear expressions via heuristics. One example is `strlen` function in libc where the operations are optimized with bit masking (example implementation for Red Hat Linux in Appendix B). This introduces bit operation instructions such as *xor* or *and*, which is non-linear operation. However, the formulas generated can be easily pattern matched once we learn the optimization used for such functions. The following is a set of formulas generated by the code in Appendix B.

```
~(0:u32 == (concat:[symb-argv_1_8:u8][concat:[symb-argv_1_7:u8][concat
   :[symb-argv_1_6:u8][ symb-argv_1_5:u8]]] - 0x1010101:u32 & 0
   x80808080:u32))
~(0:u32 == (concat:[symb-argv_1_4:u8][concat:[symb-argv_1_3:u8][concat
   :[symb-argv_1_2:u8][ symb-argv_1_1:u8]]] - 0x1010101:u32 & 0
   x80808080:u32))
```

Listing 4.2: Example of non-linear formula

4.2 Linear Inequalities and $\mathbb{Z}$-Polytope

The system of linear inequalities with the same variables can be expressed as matrix inequality $A{\cdot}x \leq b$, where A is an m-by-n coefficient matrix, x is n-by-1 variable vector and b is m-by-1 constant vector. Then, this system of linear inequalities (or *constraint system* in PPL) can directly be used to create a polytope object. Polytope object in PPL requires *non-strict* inequalities [2], so we first transform all linear expressions to be non-strict.

Sample Mayhem's formula output looks like the following:
`~(symb-argv_1_2:u8 <= 0x13:u8) & symb-argv_2_0:u8 <= 0x28:u8`, which is transformed into a set of constraints consisting PPL linear expressions:

$$Constraints\ \mathsf{cs} = \begin{cases} 1 \cdot \mathsf{Variable(0)} \geq \mathsf{0x14} \\ 1 \cdot \mathsf{Variable(0)} \leq \mathsf{0xFF} \\ 1 \cdot \mathsf{Variable(1)} \leq \mathsf{0x28} \\ 1 \cdot \mathsf{Variable(1)} \geq \mathsf{0x00} \end{cases},$$

where $\mathsf{Variable(0)}$ and $\mathsf{Variable(1)}$ map to `symb-argv_1_2` and `symb-argv_2_0`, respectively.

Since each executor invokes the *Input generator* when terminating, the module only needs to handle one path (formula) at a time. For each formula, the module extracts the range information per symbolic variable using the rules listed in Table 4.1. Note that it also includes that converts a strict inequality to a non-strict one. This rule set can easily be extended to accommodate more complex formulas that we do not support currently, or can be used to match non-linear expressions to extract linear relationship heuristically. We handle both signed and unsigned operations of the expressions (only unsigned operations shown Table 4.1, since the signed operations are analogous with different minimum and maximum) with one variable and a constant or two variables.

There are some corner cases that are not trivial to handle and lead to non-linearity or disjoint sets of linear inequalities. In our system, we mainly focus on handling these two problems that arise while transforming formula to PPL linear expression:

- **Non-linearity:** Non-linearity can come from very trivial idioms of x86 binary codes such as `xor eax, eax`. Note that this piece of code essentially is equivalent to `movl eax, 0`. Since `xor` operation is considered to be non-linear function, we cannot transform it to PPL's linear expression when the symbolic variable is involved with this type of operations. Thus, we employ transformation to convert non-linear expressions into linear expressions, when possible. This can be also used

$$\frac{\text{if } var \in \Delta \text{ then } d = \Delta[var] \text{ else } \Delta' = \Delta[var \leftarrow next\_dim] \quad d = \Delta'[var]}{\Delta, var \rightsquigarrow (d, \Delta')}\ \textsc{GetDim}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c = val}{\Delta, var == val \rightsquigarrow \Delta', \{v = c\}}\ \textsc{Equal}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c_1 = val + 1 \quad c_2 = val - 1}{\Delta, \neg\ (var == val) \rightsquigarrow \Delta', \{v \geq c_1; v \leq c_2; v \leq \mathsf{u\_max}; v \geq \mathsf{u\_min}\}}\ \textsc{NotEqual\_U}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c = val - 1}{\Delta, var < val \rightsquigarrow \Delta', \{v \leq c; v \geq \mathsf{u\_min}\}}\ \textsc{LessThan\_U}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c = val}{\Delta, var \leq val \rightsquigarrow \Delta', \{v \leq c; v \geq \mathsf{u\_min}\}}\ \textsc{LessThanEqual\_U}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c = val}{\Delta, \neg\ (var < val) \rightsquigarrow \Delta', \{v \geq c; v \leq \mathsf{u\_max}\}}\ \textsc{NotLessThan\_U}$$

$$\frac{\Delta \vdash (d, \Delta') = \textsc{GetDim}(var) \quad v = \mathsf{Variable}(d) \quad c = val + 1}{\Delta, \neg\ (var \leq val) \rightsquigarrow \Delta', \{v \geq c; v \leq \mathsf{u\_max}\}}\ \textsc{NotLessThanEqual\_U}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, var_1 == var_2 \rightsquigarrow \Delta'', \{v_1 = v_2\}}\ \textsc{EqualVar}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, \neg\ (var_1 == var_2) \rightsquigarrow \Delta'', \{v_1 \leq v_2 - 1; v_1 \geq v_2 + 1\}}\ \textsc{NotEqualVar\_U}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, var_1 < var_2 \rightsquigarrow \Delta'', \{v_1 \leq v_2 - 1; v_1 \geq \mathsf{u\_min}\}}\ \textsc{LessThanVar\_U}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, var_1 \leq var_2 \rightsquigarrow \Delta'', \{v_1 \leq v_2; v_1 \geq \mathsf{u\_min}\}}\ \textsc{LessThanEqualVar\_U}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, \neg\ (var_1 < var_2) \rightsquigarrow \Delta'', \{v_1 \geq v_2; v_1 \leq \mathsf{u\_max}\}}\ \textsc{NotLessThanVar\_U}$$

$$\frac{\Delta \vdash (d_1, \Delta') = \textsc{GetDim}(var_1) \quad v_1 = \mathsf{Variable}(d_1) \quad \Delta' \vdash (d_2, \Delta'') = \textsc{GetDim}(var_2) \quad v_2 = \mathsf{Variable}(d_2)}{\Delta, \neg\ (var_1 \leq var_2) \rightsquigarrow \Delta'', \{v_1 \geq v_2 + 1; v_1 \leq \mathsf{u\_max}\}}\ \textsc{NotLessThanEqualVar\_U}$$

Table 4.1: Mayhem expression to PPL linear expression Transformation Rules.

for transforming strict inequalities (i.e. $<$, $>$) into non-strict inequalities (i.e. $\leq$, $\geq$) as shown by LESSTHAN_U and NOTLESSTHANEQUAL_U rules in Table 4.1.

- **Disjoint sets:** Disjoint sets of linear inequalities also pose a problem of exponential blow-up in the number of constraint systems and thus the number of polyhedra to keep track of. Since all of the ranges must be represented as a linear expression with non-strict inequalities, the formula such as `x != 0` needs to be split into two different inequalities, namely `x` $\leq$ `-1` and `x` $\geq$ `1`. This leaves a *hole* in the possible range that the variable is bound to. This type of formula is common in the context of checking if a certain symbolic byte is a *null* byte. Also, whenever we have the equality in a formula such as `x == 10`, there always exists one or more formula that contains $\neg$`(x == 10)`, which is essentially `x != 10` that leads to another path in the program. These discrete sets of inequalities can be tracked to build all combinations of possible polyhedra of the input space as shown in Figure 4.1. Note that the number of all combinations grow very quickly and soon make it infeasible to enumerate constraint systems, build polytopes, and perform rejection sampling against all of them. In our experiment, we can only handle a formula with up to 16 variables that have discrete property (i.e. != relationship) under 10 minutes per formula.

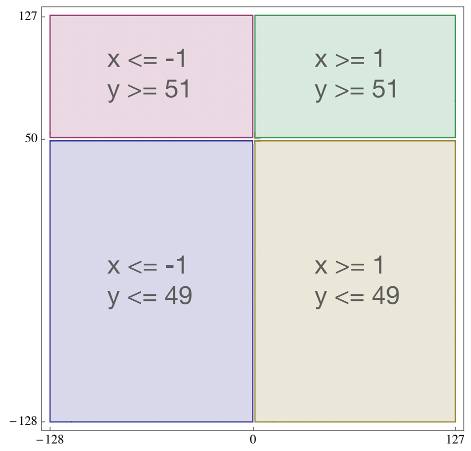

Figure 4.1: Disjoint regions described by constraints: {`x!=0 & y!=50`} for signed bytes x and y.

In order to resolve this common case without hurting the performance, we have implemented a *base shifting trick* that basically shifts the ranges of the variable by certain offsets to eliminate the hole, and thus remaining with continuous range that we can represent without multiple disjoint regions. We take the possible value right after the hole and shift it to become the minimum value (e.g. u_min or s_min) and keeps track of a mapping of the offset. Then we take advantage of modular arithmetic for the integers in computer systems. Integer wraparound behavior lets us to re-define the range for each variable. When we are actually picking a value for a certain variable, the module first picks a random value from the refined range and it looks up the offset table to obtain appropriate offset. The final value, then, is simply calculated by adding the offset to the random value the module just picked. Our shifting algorithm guarantees that only the values in the valid range (i.e. two original discrete ranges) are chosen. Note that this trick works only when there is one gap in the range, whether it is one value or a range of the values. The refinement algorithm is presented in Algorithm 1 below.

Algorithm 1: Constraint Refinement with Base Shifting

```
Input: (Discrete) Constraints CS for variable $v$
Data: OffsetTable OT, $d$, $start$, $end$, $min$, $max$, $offset$
Output: Constraints CS', OffsetTable OT'
begin
    $d \leftarrow$ getDimension($v$)
    foreach $c \in$ CS do
        if isLessEqual($c$) then
            $start \leftarrow$ getUpperBound($c$)
        end
        if isGreaterEqual($c$) then
            $end \leftarrow$ getLowerBound($c$)
        end
    end
    if isSignedByte($v$) then
        $min \leftarrow$ s_min
        $max \leftarrow$ (s_max $+ start - end + 1$)
    else
        $min \leftarrow$ u_min
        $max \leftarrow$ (u_max $+ start - end + 1$)
    end
    $offset \leftarrow end - min$
    OT' $\leftarrow$ OT[$v \leftarrow offset$]
    CS' $\leftarrow$ {Variable(d) $\geq min$; Variable(d) $\leq max$}
    return CS', OT'
end
```

Consider the case where we want to deal with the range of the valid values described by `x !=10` with unsigned byte variable `x`. Then, this essentially means the two valid regions are {u_min $\leq$ `x` $\leq$ `9`} and {u_max $\geq$ `x` $\geq$ `11`}. This, in a diagram, can be shown as on the left side of Figure 4.2 which shows clear two disjoint

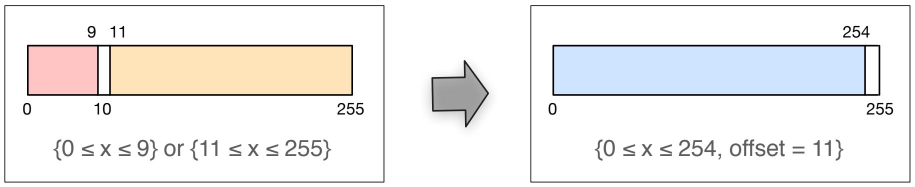

Figure 4.2: Constraint Refinement of `x!=10`

regions. However, after applying the shift, we get a new range and a new entry in the offset table. The effect of the shifting is shown on the right side of Figure 4.2. Note that the new range is a single continuous block that we can easily deal with PPL in order to build the polytope and to pick a random point quickly for testing. In a way, we are hiding true representation of certain variables in order to make the problem simpler and adjust it accordingly by consulting the table we have built in the process. The same technique is equally applicable to signed integers without any modification. Example of signed byte with larger gap is shown in Figure 4.3.

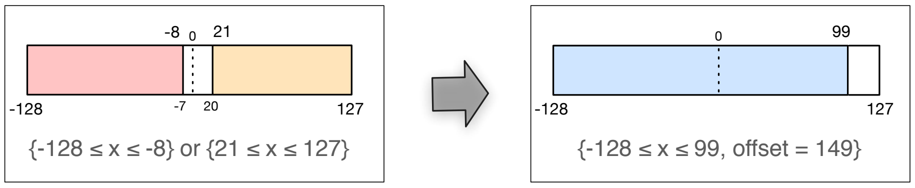
-8
0
21
-128
-7
20
127
{-128 ≤ x ≤ -8} or {21 ≤ x ≤ 127}
0
99
-128
127
{-128 ≤ x ≤ 99, offset = 149}

Figure 4.3: Constraint Refinement of $\neg(-7 \leq x \leq 20)$

Apart from the transformability, we also take a careful approach in handling both signed and unsigned operations. Since the sign-ness determines the minimum and maximum values, we need to take the sign-ness of the operation into account to first deduce the type of the symbolic byte (i.e. signed char vs. unsigned char) and then apply appropriate extreme values such that we correctly choose the random point that indeed will satisfy the original formula. The difference between signed byte and unsigned byte is shown in Figure 4.4. The same concept applies to larger data types such as `short int` or `long int`.

When building test cases, we sometimes need to avoid certain values for symbolic bytes even though it is allowed from the extracted range. For example, we need to avoid choosing string terminators or special characters that have specific effect when used in program arguments or environments such as a space (`0x20`), tab (`0x09`), and new line (`0x0a`). In order to make this easily, we have made an option to include *bad character*

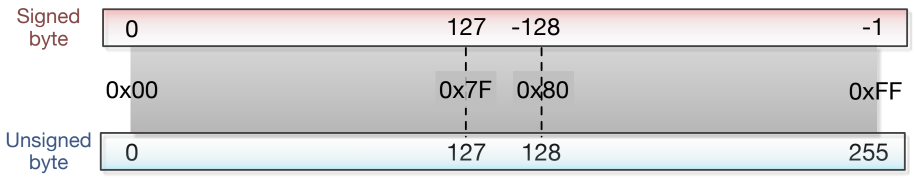

Figure 4.4: Range of byte value depending on the sign-ness

set, which will be later avoided as a value when picking a random point. If it is not possible to pick a value without picking from the values specified *bad character set*, then the error is raised.

Relationships between byte-level symbolic variables stay quite simple since we can treat each symbolic variable as a PPL variable in a distinct dimension. However, if there is a multi-byte (e.g. 32-bit integer) operation in the formula, the expression is represented as the concatenation of 1 byte (8-bit) characters. It is much harder to extract separate range definitions for each byte in this case. Consider the following example of 32-bit unsigned integer comparison:

`concat:[b_3:u8][concat:[b_2:u8][concat:[b_1:u8][b_0:u8]]] < 0x41424344:u32`

Note that the ranges of the values for each byte is dependent of *more-significant-byte*. In the example above, the value of `b_3` can be expressed as a following inequality: `b_3` $\leq$ `0x41`. Then, two cases are introduced for determining the value of `b_2`. First is when `b_3 = 0x41` where `b_2` is restricted to be `b_2` $\leq$ `0x42` to satisfy the original inequality, propagating similarly to *lesser-significant-bytes*. However, in case of `b_3 < 0x41`, the remaining lesser-significant-bytes `b_2`, `b_1` and `b_0` are free in the range of the unsigned minimum (i.e. 0) and maximum (i.e. 255), because `b_3 < 0x41` already guarantees the original inequality to hold. In order to handle this issue, rather than introducing the disjunction of possible combinations of the ranges, we treat it as a single linear inequality that is expressed as $b\_3 \cdot 2^{24} + b\_2 \cdot 2^{16} + b\_1 \cdot 2^{8} + b\_0 \cdot 2^{0} \leq \text{0x41424344}$. Obviously , this approach can be generalized to incorporate data type of any size, such as 16-bit `short int` or 64-bit `long long int`.

4.3 Hypercube and Rejection Sampling

Since efficiently enumerating all integer points in $\mathbb{Z}$-Polytope that describes input space in n dimension is not feasible, we propose an approximation of the space by defining the bounding box (*hyperrectangle*) that inscribes the polytope resulting an over approximation of the region. Finding the boundaries of the hypercube is trivial in our setting, so it is easy to select random points that reside in such hyperrectangle. However, we need to verify that the chosen point is indeed in the original polytope that describes valid input space. For this we perform a *rejection sampling*.

4.3.1 Hypercube

In case of different length of sides per dimension, the term *hyperrectangle* is used, but we will use both terms interchangeably in our context as we discussed in previous chapter. Note that the sides of the surrounding box can always be forced to be of the same length, but it is better to obtain the tightest length of the sides possible in order to reduce the gap between the actual polytope and the face of the hypercube in any dimension. This is because the probability of finding a valid random point from the hypercube increases as the volume between the polytope's surface and the hypercube's boundaries decreases. Thus, this often makes our hypercube in rectangular shape.

Consider an example of the input space and surrounding hypercube (in this case, it's 2-dimensional rectangle) shown in Figure 4.5. In this figure, the input domain is specified by the following inequalities, which is derived from the formula:

$$\{symb\_0 \leq 90\} \wedge \{symb\_1 \leq symb\_0\} \wedge \{symb\_1 \geq \tfrac{symb\_0}{4}\}$$

The intuition on building hypercube is easily seen in that we have much higher probability to pick a point that resides in the actual input space if we choose a random point from the hypercube instead of the universe. Note that the area (or volume depending on the dimension) of the universe expands extremely fast as the number of dimension grows. Specifically, the size of the universe is 256^d, where d is the number of dimensions for the input space. For simplicity, we assume that each dimension represents one byte, which represents $2^8 = 256$ integer points. We reduce the dimension for the polytope by limiting the scope of variables only to path critical symbolic bytes.

Lower bound and Upper bound In order to pick a random point inside the hypercube efficiently, we need to find out the boundaries (lower and upper bounds) of the cube for each dimension. Unlike polytope, it is easy to compute the lower bound and upper bound of the hypercube that inscribes the polytope which represents the precise input space. We use linear programming to obtain both minimum and maximum values for each dimension (variable). Linear programming is a method for the problem of achieving the maximum or the minimum of a linear function in a given set of constraints. PPL already provides

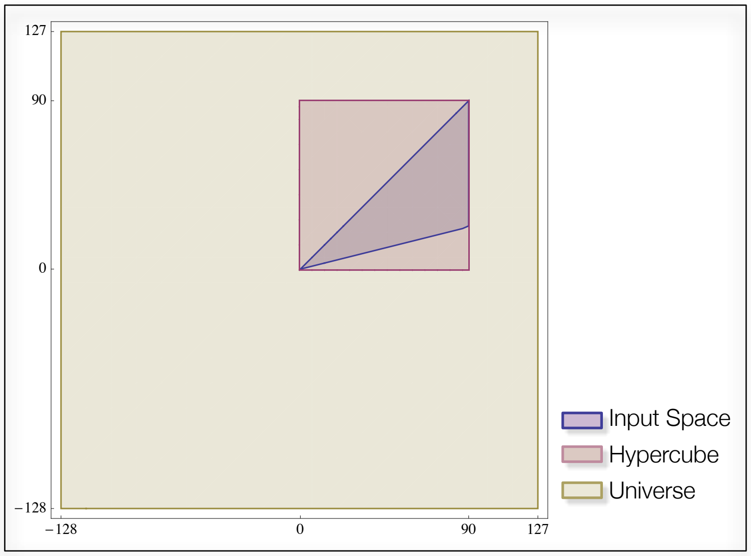

Figure 4.5: Visualization of the input space and hypercube for *symb_0* and *symb_1*.

methods to maximize and minimize the given linear objective function subject to the constraint system for the polytope. By maximizing and minimizing an objective function that only consists of a single variable, we can get the upper bound and the lower bound of the variable, respectively.

For example, if we want to find the boundaries for the hypercube that tightly inscribes the input space (a blue triangle) in Figure 4.5, we can first minimize and maximize the variable *symb_0* and repeat for the variable *symb_1*. Precisely, we get the following result, producing the purple square we observe above.

$$symb\_0 = \begin{cases} min : 0 \\ max : 90 \end{cases}, \quad symb\_1 = \begin{cases} min : 0 \\ max : 90 \end{cases}$$

Notice that the generated hypercube still has some gaps between its boundaries and the input space polytope. It is, however, far smaller compared to the gap from the universe. We basically reduced the domain we choose the random points from, and thus significantly increased the probability to pick the points inside of the polytope when the point is taken from the hypercube.

4.3.2 Rejection Sampling

In mathematics, *rejection sampling* is used to generate a random set of samples from a distribution by sampling uniformly from the probability density function's region. This is useful when it is difficult to sample points under the original probability distribution function. Consequently an instrumental distribution – that is easier to sample from and contains the original distribution – to perform the sampling. Then, the verification of the chosen point against the original distribution and its rejection or acceptance is determined.

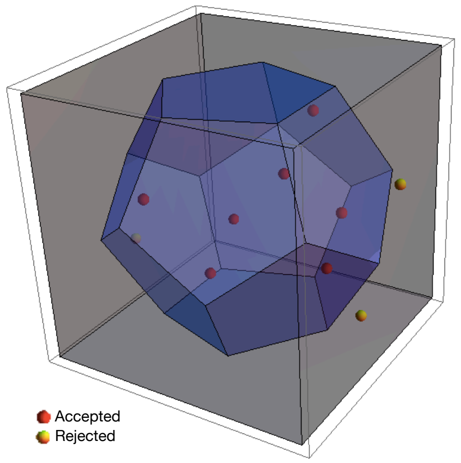

Figure 4.6: Visualization of rejection sampling.

In our setting, the original probability distribution is the original polytope of the input space and the instrumental distribution corresponds to our hypercube. The random points in the hypercube are *sampled* and tested for the acceptance based on relationship between the polytope – i.e., the point is accepted if it resides in the polytope, but rejected otherwise. As shown in Figure 4.6, points are chosen from the cube (3-hypercube), then the points that reside inside of the polytope are accepted whereas the ones that are outside are rejected.

Random Sampling We can also use an extended rejection sampling, such as adaptive rejection sampling, in order to provide better and tighter hypercube boundaries by reducing wasted space (gap). However, this requires successive and adjacent point selection to refine the distribution models *adaptively*. In context of fuzz testing, random sampling is more suitable to increase the probability for making the program to exercise more various paths by providing provably random inputs.

Discontinuous Ranges and Powerset Sometimes we observe that the formula is in linear form, but it consists of one or more disjoint ranges. For instance, $\{0 \leq x \leq 10, 80 \leq x \leq 100, -128 \leq y \leq 127\}$ defines two disjoint regions: $\{0 \leq x \leq 10, -128 \leq y \leq 127\}$ and $\{80 \leq x \leq 100, -128 \leq y \leq 127\}$. In this case, it is not possible to define these inequalities as a single polytope. Another example – more commonly found – is inequality such as $\{x \neq 7\}$. Note this also splits the input space into two disjoint regions, namely $\{x < 7\}$ and $\{x > 7\}$. However, we demonstrated a method to transform these two disjoint regions into a single continuous range by refining the constraints with base shifting as explained in Section 4.2. The visualization of discontinuous ranges for example above is shown in Figure 4.7.

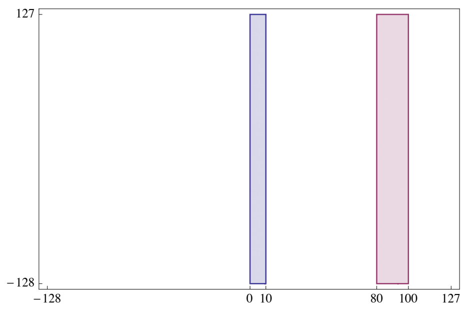

Figure 4.7: Visualization of disjoint regions.

For the correctness, we need to ensure that the actual constraints are respected when we test for rejection sampling. This means that we need to represent these types of the input space precisely without approximation. We achieve this by using a *powerset*. We build a powerset of disjoint polytopes generated by disjoint ranges. Building such powerset can easily become exponential overhead with respect to the number of disjoint regions we need to keep track. However, most of the constraints we examine were disjoint regions with only one gap (e.g. $x \neq c$ case).

Chapter 5

Evaluation

In this chapter, we evaluate our proposed approach through three different synthetic example programs and twenty different x86 Linux binary programs that Mayhem is capable of handling. In each section, we highlight the core components of our experiment and point out how different configuration affects overall results for the code coverage, which is closely related to the probability of finding bugs. Each component can be optimized individually to provide a smoother transition between phases and a better result overall. Source code for all of the synthetic program examples is attached in Appendix C.

The machine we used for all of our experiments is a desktop with Intel Core 2 CPU 6600 @ 2.40GHz, 4GB of RAM, and 32-bit Ubuntu Linux 11.10. The implementation used for the experiment is not pipe-lined and assigns random value to the symbolic bytes that the system cannot reason about due to non-linearity or unimplemented features. Although the current system implements the basic block profiling, we have not used this feature in the experiment due to lack of initial seed inputs and time. We discuss on possible improvements in Chapter 6.

5.1 Formula Transformation

We focus our evaluation on the *transformability* of the formulas that are generated in `Symbolic Execution` phase. Recall that the transformability means the *linearity* of the formula as well as the capability of our transformation rules. Transform is successful if the resulting expression is in the form of linear inequalities and precise (i.e. correct range). Note that the equality, $x == a$, can be represented as a conjunction of two linear inequalities, $x <= a$ and $x >= a$. Formulas that contain non-linear relationship are out of scope of this paper, and thus not transformed. We have tested 3 synthetic programs, 15 coreutils programs and 5 other x86 Linux programs to survey how many of the total formulas generated are *transformable*.

Application	# Total Preds.	# Transformed Preds.	Transformability (%)
simple	9	9	100.0
simple_file	5	5	100.0
int_cmp	5	5	100.0
true	87	87	100.0
printenv	30110	29122	96.72
basename*	9891	6418	64.89
dirname	458	393	85.81
hostid	21394	21021	98.26
id*	17420	15944	91.53
whoami*	375312	311710	83.05
mktemp*	24635	22194	90.09
sync	6127	4324	70.56
link	1552	1340	86.34
mkdir*	4768	4606	96.60
readlink	17280	17044	98.63
ls*	9162	5489	59.91
who*	268211	206193	76.88
ptx*	592363	386634	65.27
Total	1378770	1032519	74.89

Total does not include synthetic programs.
∗ - process killed after 60min.

Table 5.1: Predicate Transformability in synthetic programs + coreutils

Application	# Total Preds.	# Transformed Preds.	Transformability (%)
htpasswd-1.3.31*	2929	2850	97.30
cdescent-0.0.1	70787	70742	99.94
fkey-0.1.3	7038	4301	61.11
exim-4.41*	40122	39555	98.59
sharutils-4.2.1*	680	398	58.53
Total	121556	117846	96.94

∗ - process killed after 60 min.

Table 5.2: Predicate Transformability in linux binaries

In the experiment of 15 coreutils programs, we have successfully transformed 74.89% of the total predicates into PPL linear expressions as shown in Table 5.1. A *predicate* in this experiment is the smallest formula without any conjunction. Thus, this result shows that most of the predicates that are generated from the branch conditions are linear and simple. The majority of the *unhandled* formulas are due to the non-linear operations such as *and*, *xor*, and *not*. We currently do not support bit shift operations, but these are still linear operations and can be implemented by encoding them to either multiplication or division by 2^b where b is the number of bits shifted. The remaining predicates either consist only with non-linear operations or mix of both linear and non-linear operations. The transformability shown in Table 5.2 for 5 Linux utilities also show that most (96.94%) of the predicates we encounter are linear and handled by the current system.

Note that, however, this does not necessarily imply that we successfully transform the entire path with the probability demonstrated in above tables. As shown in Table 5.3 and Table 5.4, the transformability for full path is lower. This is because a path consists of multiple predicates, where each predicate may not necessarily be applicable for the transformation. If any part of the predicate transformation fails, then the remaining predicates in corresponding path are meaningless since we cannot guarantee the satisfiability of the predicate that we failed to transform. In order to overcome this limitation, we can invoke the solver to achieve a possible solution for such predicate and assign the returned values appropriately. However, we have not implemented this extension in the current stage of the research. If this component is implemented, we can expect to have similar transformability of the paths to one of the predicates.

The following is a sample path formula that is not handled by the current system:

```
~(low:u8((extend:u32(symb-argv_1_1:u8) & 0xffff00ff:u32 | pad:u32(low:u8(extend:u32(symb-argv_1_1:u8))) << 8:u32) << 0
     x10:u32 & 0xffff0000:u32 | pad:u32(low:u16(extend:u32(symb-argv_1_1:u8) & 0xffff00ff:u32 | pad:u32(low:u8(extend:
     u32(symb-argv_1_1:u8))) << 8:u32))) == low:u8((extend:u32(symb-argv_1_1:u8) & 0xffff00ff:u32 | pad:u32(low:u8(
     extend:u32(symb-argv_1_1:u8))) << 8:u32) & 0xffffff00:u32))
```

The most important message to take away from these results is that we can *significantly* reduce the number of invocation to the constraint solver for generating inputs – especially for many random path-preserving test cases, which is a huge improvement in performance compared to traditional methods.

Application	**# Total Paths**	**# Transformed Paths**	**Transformability (%)**
simple	3	3	100.0
simple_file	3	3	100.0
int_cmp	3	3	100.0
true	14	14	100.0
printenv	1151	1067	92.70
basename*	298	35	11.74
dirname	38	27	71.05
hostid	702	633	90.17
id*	416	43	10.34
whoami*	10018	2256	22.52
mktemp*	902	3	0.33
sync	193	3	1.55
link	87	45	51.72
mkdir*	103	56	54.37
readlink	661	598	90.47
ls*	276	9	3.26
who*	3495	6	0.17
ptx*	10836	4	0.04
Total	29190	4799	16.44

Total does not include synthetic programs.
∗ - process killed after 60min.

Table 5.3: Path Transformability in synthetic programs + coreutils

Application	**# Total Paths**	**# Transformed Paths**	**Transformability (%)**
htpasswd-1.3.31*	172	153	88.95
cdescent-0.0.1	372	327	87.90
fkey-0.1.3*	250	7	2.80
exim-4.41	377	101	26.79
sharutils-4.2.1*	34	3	8.82
Total	1205	591	49.05

∗ - process killed after 60 min.

Table 5.4: Path Transformability in linux binaries

5.2 Input Generation

Efficient and effective input generation is important to both developers and security researchers. Generated inputs can be tested against the target program to be monitored for any abnormal behaviors such as crash. Symbolic execution is usually capable of generating an input to a certain path by querying the constraint solvers – the solution which satisfies the formula is the input itself. However, it may not reveal other significant portion of possible inputs in case there are ranges of values that the input bytes could have.

With the assistance of $\mathbb{Z}$-Polytope abstraction, the hybrid fuzzing system can derive and produce a large number of random inputs that will respect the path constraints. The random value will be chosen for each symbolic input byte within the range of the values that the byte can have without disrupting the satisfiability of the given formula. This is critical since generating such inputs require many queries to the solver, which causes high latency. We generate N input samples per path for P frontier nodes (paths) as configured by the user, generating total of $N \times P$ random inputs.

Application	Zzuf Gen. Time (sec)	Hybrid Fuzzer Gen. Time (sec)	# Test Cases
true	9.6	4.5	7000
printenv	68.6	48.0	25000
basename	29.6	41.0	11000
dirname	51.9	33.5	19000
hostid	73.2	27.3	25000
id	78.0	84.1	25000
whoami	76.5	19.7	25000
mktemp	76.3	87.6	25000
sync	74.1	59.1	25000
link	73.2	34.9	25000
mkdir	73.0	23.6	25000
readlink	77.2	23.8	25000
ls	65.7	26.4	25000
who	63.2	66.8	25000
ptx	63.6	82.9	25000
Total	953.7	663.2	337000

Total of 500 test cases generated per frontier node per application (max 50 frontier nodes).
The time is averaged over 5 independent runs.

Table 5.5: Statistics on Input Generation

Application	Zzuf Gen. Time (sec)	Hybrid Fuzzer Gen. Time (sec)	# Test Cases
htpasswd-1.3.31	80.0	9.2	25000
cdescent-0.0.1	64.5	26.4	25000
fkey-0.1.3	56.4	33.3	25000
exim-4.41	90.2	24.6	25000
sharutils-4.2.1	55.4	136.1	25000
Total	346.5	229.6	125000

Total of 500 test cases generated per frontier node per application (max 50 frontier nodes).
The time is averaged over 5 independent runs.

Table 5.6: Statistics on Input Generation

We present the generation time that hybrid fuzzer took to generate 500 random inputs per path in various different programs in Table 5.5 and Table 5.6. For practicality, we have limited the maximum number of frontier nodes to 50 in this experiment. This means 50 unique paths were symbolically executed after the symbolic inputs are introduced to the target programs, which is usually sufficient in the context of argument parsing and simple file parsing.

Statistics show that, in both coreutils and other Linux utilities, the hybrid fuzzer generates the same number of inputs about 45% faster than the mutational fuzzer does in average. This is because the mutational fuzzer needs to open up the seed file, create new (fuzzed) file, modify the values given the ratio, and finally write to disk, whereas the hybrid fuzzer only creates and writes to the new file with already (randomly) chosen values. We have not tested other mutational fuzzer than Zzuf, but it may be possible to optimize the mutation process to be faster.

An experiment that we could not perform due to the time limit is the input generation time by the symbolic executor with constraint solver. Although we did not have a chance to provide data, our hypothesis is that generating the same number of test cases using constraint solver would take a lot longer time compared to both mutational fuzzer and hybrid fuzzer. The reason is that, in order for the symbolic executor to create distinct and random test cases, it needs to invoke the constraint solver for finding an assignment that satisfies the given formula and is different from previously generated one. This requires one or more invocations to the constraint solver each time the system needs to generate a new test case, which is expensive.

Generated test cases are also tested for the uniformity of randomness. We have generated 50,000 test cases per path in synthetic program *simple* (C.1) and plotted the distributions per path in Figure 5.1. It shows a particular range of values that has been assigned more or less in each path. This coincides with the actual code: Third byte in `argv[1]` (*b_0*) needs to be greater than or equal to 20 (line 8) and the first byte in `argv[2]` (*b_1*) needs to be less than or equal to -80 (line 10) simultaneously in order to reach `PATH 1`. For `PATH 2`, *b_0* needs to be greater than or equal to 20, but *b_1* needs to be greater than -80. Lastly, *b_0* must be be less than 20 in order to reach `PATH 3`, regardless of the values for any other input bytes. Note that since the input for *simple* is program arguments, we specified some bad characters (e.g. null character, tab, new line, etc.) which is shown by the byte values of 0 frequency. Within the valid ranges of the input, however, the values are uniformly distributed.

5.3 Code Coverage

Code coverage is one of the standard metrics in program analysis because it represents how much of the code is exercised. Higher code coverage does not necessarily imply more bugs to find, but it definitely increases the probability of finding them. By comparing our result against traditional random mutation fuzzing and symbolic execution techniques, we show that the hybrid fuzzing is effective in achieving high code coverage in relatively shorter time period. We used *gcov* for the code coverage measurement. Specifically, the code coverage is measured in the number of lines that are executed over the total number of

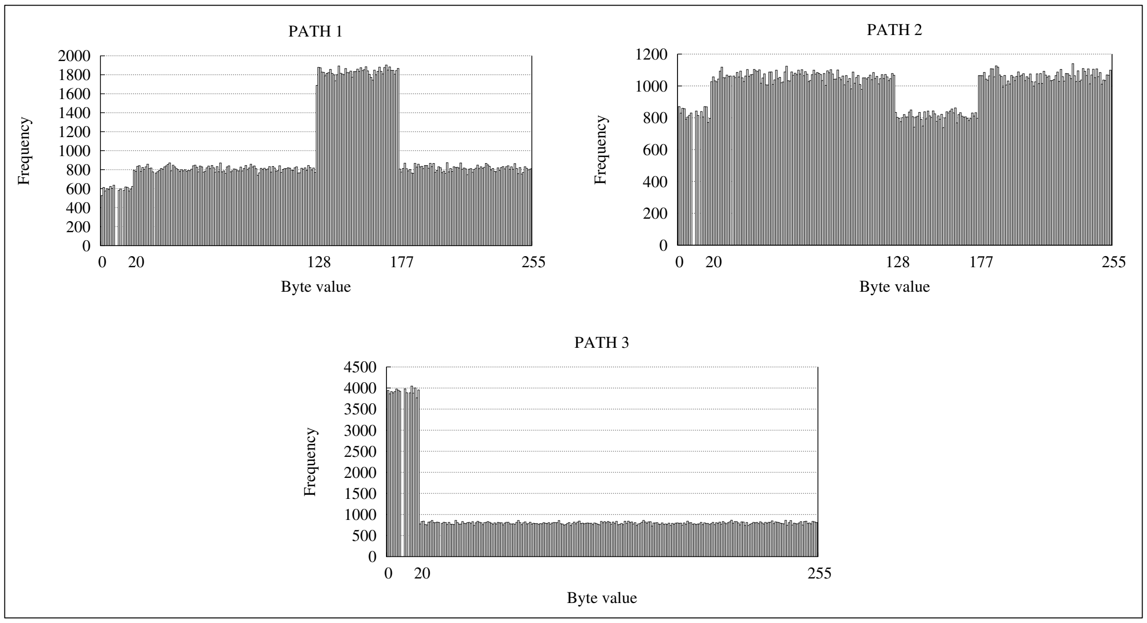

Figure 5.1: Uniform distribution of random test cases for *simple*

lines of the source code present in the target application.

vs. Random mutational fuzzing We chose *Zzuf* application fuzzer for the random mutational fuzzer [27]. Zzuf is a very fast and simple fuzzer that modifies the user-controllable data (input arguments or files) that the target application handles. It has found bugs in real-world programs and libraries such as VLC player, MPlayer, nm, and firefox [28]. In our experiment, we target 3 synthetic programs, 15 coreutils programs and 5 other Linux utilities as listed in Table 5.7 and Table 5.8. These applications are chosen accordingly to demonstrate the effectiveness in small (tens of lines), medium (hundreds of lines), and large (thousands of lines) code bases.

For the comparison, we used the same test cases that are generated in input generation experiment, which consists of $N \times P$ where N is 500 and P is at most 50. We let Zzuf to generate the same amount of the test cases by mutating the seed file $N \times P$ times for each program. Then, we measured the code coverage for feeding the test cases to the target program. Specifically, an input is randomly mutated with 20% mutation ratio with incremental seed number for creating test cases with Zzuf, and the random bytes are chosen within the valid range for the hybrid fuzzer. In case of non-linear predicate,

Application	Zzuf Code Coverage (%)	Hybrid Fuzzer Code Coverage (%)	Total # Lines of Code
simple	100.0	100.0	11
simple_file	90.0	100.0	20
int_cmp	100.0	100.0	11
true	52.63	57.89	19
printenv	73.68	78.95	38
basename	81.58	81.58	38
dirname	68.57	82.55	35
hostid	61.54	76.92	26
id	23.58	34.96	123
whoami	59.26	85.19	27
mktemp	62.11	66.32	95
sync	80.95	85.71	21
link	71.88	84.37	32
mkdir	48.53	69.12	68
readlink	50.00	60.42	48
ls	22.43	34.51	1382
who	24.15	49.81	265
ptx	43.18	43.48	660
Total	33.68	44.03	2877

Total does not include synthetic programs.

Table 5.7: Code coverage for synthetic + coreutils (Fuzzing)

Application	Zzuf Code Coverage (%)	Hybrid Fuzzer Code Coverage (%)	Total # Lines of Code
fkey-0.1.3	19.51	25.20	123
htpasswd-1.3.31	38.71	70.05	217
cdescent-0.0.1	34.22	36.40	1011
sharutils-4.2.1	17.66	28.28	640
exim-4.41	8.78	11.90	5055
Total	14.35	18.93	7046

Code that is not reached by both fuzzers are not counted.

Table 5.8: Code coverage for linux binaries (Fuzzing)

we currently assign random values with full flexibility (i.e. 0-255 for a byte) in the hope of satisfying the condition sometimes. If there are other linear predicates that provides a tighter bound for a certain byte, the system prefers the tighter bound to the wider bound by design (when building the polytope). In the worst case, where all of the predicates are non-linear, hybrid fuzzer system defaults to a random fuzzer in current implementation.

As shown in Table 5.7 and Table 5.8, the hybrid fuzzer mostly covers more code than random mutational fuzzer does. Because most of the predicates in each path are linear as we discovered earlier, the hybrid fuzzer can reason about the precise input space (ranges) in order to generate test cases that will lead to different paths.

vs. Symbolic execution We used *Mayhem* binary symbolic executor [40] to explore as many paths as possible and generate test cases with the path predicates for 5 minutes. Similarly, we let the hybrid fuzzer run and generate 200 fuzzed test cases per frontier node it was able to reach in 5 minutes. Note that the time restriction includes the time to generate inputs as well. Furthermore, most of the paths in the programs are discovered in first 5 minutes with symbolic execution. However, the number of paths and the elapsed time can highly vary depending on the symbolic execution configuration. To conduct fair comparison, we used the same configuration for the symbolic execution regarding what and how long to make symbolic.

We let both hybrid fuzzer and symbolic executor explore and generate test cases in a given time (which is 5 minutes each), and measured the code coverage for executing the target programs with generated test cases. Hybrid fuzzer is set to create 200 random test

Application	Mayhem Code Coverage (%)	Hybrid Fuzzer Code Coverage (%)	Total # Lines of Code
simple	100.0	100.0	11
simple_file	100.0	100.0	20
int_cmp	100.0	100.0	11
true	52.63	57.89	19
printenv	100.0	78.95	38
basename	65.79	81.58	38
dirname	100.0	80.00	35
hostid	100.0	69.23	26
id	56.10	22.76	123
whoami	85.19	85.19	27
mktemp	54.74	66.32	95
sync	66.67	85.71	21
link	96.88	81.25	32
mkdir	70.59	73.53	68
readlink	52.08	66.67	48
ls	45.22	37.99	1382
who	84.53	49.81	265
ptx	63.18	43.18	660
Total	57.77	45.18	2877

Total does not include synthetic programs.

Table 5.9: Code coverage for synthetic + coreutils (Symb)

Application	Mayhem Code Coverage (%)	Hybrid Fuzzer Code Coverage (%)	Total # Lines of Code
fkey-0.1.3	23.58	23.58	123
htpasswd-1.3.31	77.42	76.04	217
cdescent-0.0.1	45.50	53.41	1011
sharutils-4.2.1	17.81	33.13	640
exim-4.41	17.10	17.34	5055
Total	23.92	25.86	7046

Table 5.10: Code coverage for linux binaries (Symb)

cases per frontier node in this experiment.

Although the code coverage varies depending on the applications and the configuration, we can observe that symbolic executor performs better in overall. Interesting to note here is the types of the target programs that hybrid fuzzer performs better such as *basename*, *mktemp*, and *readlink*, where the symbolic execution does not terminate within 60 minutes with the given configuration. It shows that hybrid fuzzer can be effective with the targets that have characteristics of yielding slow symbolic execution. Since the programs in coreutils are rather small in size and simple, the symbolic executor performs quite well even in 5 minutes. However, with more complicated or larger programs as listed in Table 5.10, the hybrid fuzzer discovers more code than the symbolic executor does in first 5 minutes.

Again, given enough time and resource, traditional symbolic executor will outperform the current hybrid fuzzer due to an inability to handle non-linear predicates. This can, however, be improved by integrating the constraint solver to the system which we discuss in Chapter 6.

5.4 Observation

In this section, we present some statistics on different configurations such as the number of generated random test cases and the time restriction in symbolic execution phase. Also, we construct and explain graphs that show the change in the code coverage over time as well as the number of test case generated. The results highlight the speed and the effectiveness of the hybrid fuzzing technique compared to traditional ones.

Code coverage vs. number of generated inputs Part of the strength in our hybrid fuzzing technique comes from its randomness. Leveraging the flexibility in the input space by capturing the precise range for each input byte enables us to achieve higher probability to reach the code paths that `Symbolic Execution` phase has not explored in a given time. Therefore, in this experiment, we expect to see that the more random path-preserving test cases we generate, the higher code coverage we will reach especially when there are not much time to explore all the paths.

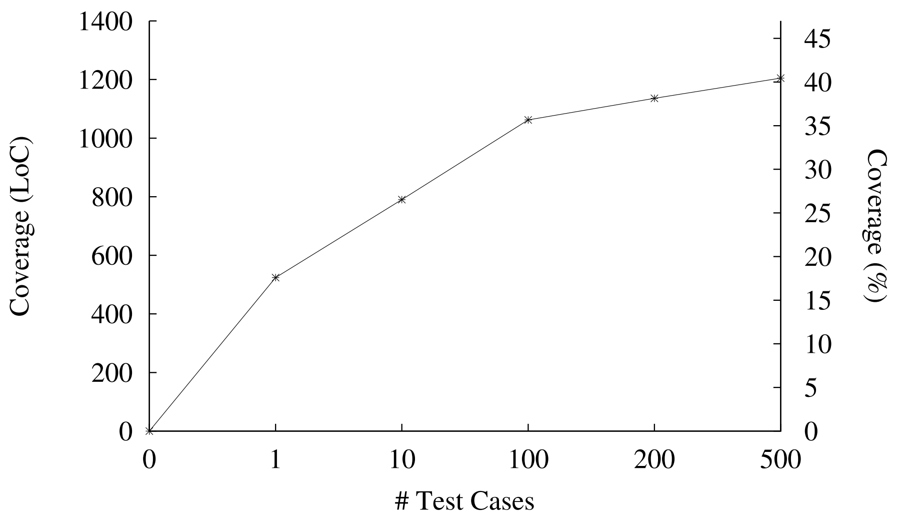

Figure 5.2: # Test Cases vs. Code Coverage

We measured the code coverage over different number of test cases generated. Specifically, we restricted the execution time to 1 minute and created the following number of test cases: 1, 10, 100, 200, and 500 for 6 selected applications based on their size. Specifically, we chose *printenv, mktemp, ls, who, htpasswd-1.3.31*, and *cdescent-0.0.1*. As shown in Figure 5.2, the total code coverage increases as more test cases are generated. This confirms our belief that the randomness (introduced by generating many path-preserving uniformly random test cases) indeed helps augmenting the code coverage.

Code coverage over time In this experiment, we have most of the configuration as same as above, where we run Zzuf fuzzer to generate as many test cases as possible by randomly mutating the given input, allow Mayhem symbolic executor to explore as many paths as possible to generate one input per unique path, and finally run Hybrid fuzzer to explore and generate 200 guided random inputs per path, all within the given time period. Then, we execute the target programs with test cases that are generated by each technique to measure the code coverage. Note that we have randomly chosen the values for the bytes that are associated with non-linear constraints (unless there were another linear constraint that restricts the range of them), introducing to some non-path-preserving test cases.

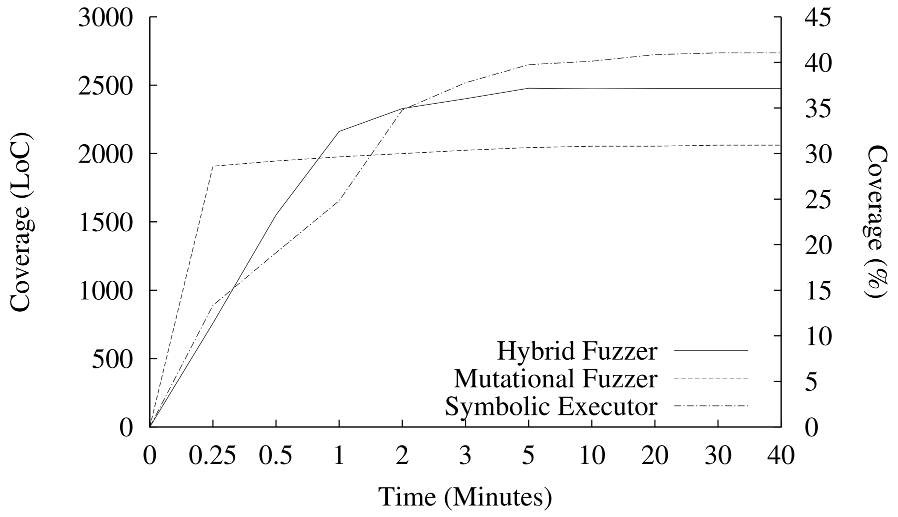

Figure 5.3: Time vs. Coverage for coreutils + other Linux utilities (20 total)

The graph in Figure 5.3 confirms the traits of each technique as we expected. Code coverage from fuzzing spikes up really high in first 30 seconds, but quickly saturates onward such that it barely increases for the rest of the period. Both symbolic execution and hybrid fuzzer increase the code coverage slowly in the beginning and eventually achieve higher coverage than mutational fuzzer's. When the permitted time is relatively short (20 to 90 seconds in Figure 5.3), the hybrid fuzzer acquires about 25% higher code coverage compared to Mayhem symbolic executor. This time frame, also marked by the region between three curves below the hybrid curve, is when the hybrid fuzzer outperforms other methods. After couple minutes, the symbolic executor catches up and discovers more paths than the hybrid fuzzer. The code coverage for both symbolic executor and hybrid fuzzer saturates soon after or finds more distinct paths very slowly depending on the target application and the corresponding configuration.

Theoretically, the hybrid fuzzer should take the lead from the start until the symbolic executor takes over. However, we observe that the symbolic executor actually performs better in the first 15 seconds or so. This is because of the non-linear predicates that the system face in the beginning of `Symbolic Execution` phase. When the symbolic bytes are not constrained too much in the beginning of the execution, non-linearity of the predicates lead the hybrid fuzzer to generate merely random test cases rather than guided ones, whereas the symbolic executor invokes the constraint solver which guarantees the execution of the paths that it has found so far. We present the distribution of the linearity of the path predicates we encounter for *mktemp* program in Figure 5.4. As the figure shows, few non-linear predicates (marked as red) appear in the beginning of the path predicates and the linear predicates (marked as gray) follow for the remaining part of the path. This means that we are blocked early with non-linear constraints that we do not handle currently and the system act as a random input generator more or less. That is why we see the code coverage by the hybrid fuzzer overtakes one by the symbolic execution after 20 seconds in Figure 5.3. If we handle the non-linear constraints that are directly followed by the first linear constraint, we can achieve high path transformability close to the individual predicate transformability as shown in Table 5.1, which will significantly increase the code coverage by guaranteeing each path more precisely.

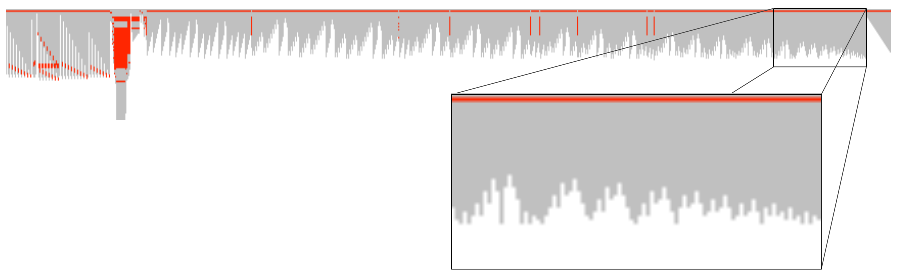

Figure 5.4: Non-linear vs. Linear Predicates in *mktemp*

Also, with the same reason, the hybrid fuzzer does not reach the code coverage of the symbolic executor. Some predicates in the path predicates are very constrained, non-linear and complex, which the current system cannot handle only with PPL abstraction. It would be an interesting and necessary future work to extend the current design to integrate the constraint solver for non-linear predicates and observe how it performs in discovering code. Our hypothesis is that the extended system will provide a lot faster code discovery with less resource (e.g. less constraint solver invocation).

There are some cases that the hybrid fuzzer dominates both the mutational fuzzer and the symbolic executor from the beginning (at least until the symbolic executor takes over), such as *htpasswd-1.3.31* shown in Figure 5.5. With this application, we see high path transformability and simple formulas (generally plain equality of a byte), allowing the hybrid fuzzer to perform extremely well from the beginning. It is clear that the region where the users can get benefits from our system over other techniques has significantly larger in this example.

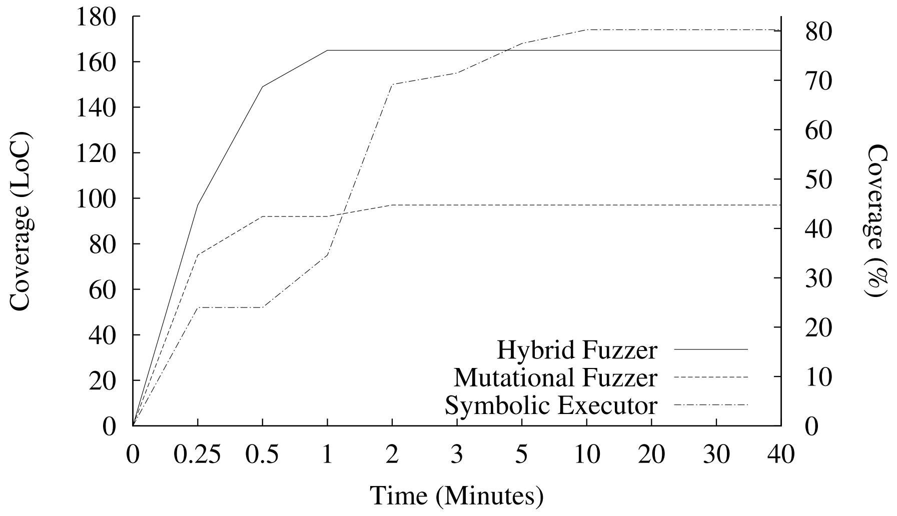

Figure 5.5: Time vs. Coverage for *htpasswd-1.3.31*

Chapter 6

Discussion

In this chapter, we discuss possible limitations and improvements in our hybrid fuzzing system. We describe known issues and explain how each component can be improved and optimized in order to make a more robust and effective fuzzing system. We also discuss about various configurations that we were not able to experiment and their implications towards our system. Lastly, we talk about future works that can be built upon the hybrid system.

6.1 Discussion

6.1.1 Limitations

There are few limitations that our hybrid fuzzing system has due to either the underlying design of the system. We discuss the limitations we briefly mentioned in earlier chapters and suggest possible solutions.

Non-linear formulas. Hybrid fuzzer is solely using an abstraction of $\mathbb{Z}$-Polytope, which limits the scope to linear constraints. Thus, if there exist non-linear formulas in the path predicates, the system cannot extract possible ranges of the values for each symbolic byte precisely. This challenge can be overcome by invoking the theorem prover to reason about the possible values or relationships with some performance degradation. However, as we observed in Chapter 5, there are very few non-linear formulas we encounter in usual programs.

Another method to solve the problem of non-linearity is to use heuristics. When we find a non-linear formula, we can try to extract constants from it and quickly perform few satisfiability tests for each path critical variable to reason about possible range of the values. For instance, given non-linear formula `symb_0 & 0xF0` $\leq$ `0x56`, we can extract the constants `0xF0` and `0x56` to test few values that are less than, greater than, or equal to these constants satisfy the formula. We can also implement heuristics for specific non-linear operations that the behavior is well-defined and easy to over-/under-approximate, such as *and*. For the formula in above example, we can notice that the operation is essentially masking the top 4 bits and thus we can apply the masking to the right-hand-side (i.e. 0x56) and obtain refined (and linear) formula, `symb_0` $\leq$ `0x50`.

Multiple disjoint ranges. If we use the powerset of the polytopes as the current system does, we can guarantee the correctness when generating inputs. As we discussed in previous chapters multiple times, however, it is quite difficult to represent the powerset of the polytopes efficiently. Since the disjoint ranges introduce disjunctive relations, the amount of the constraint systems quickly becomes large making it infeasible to compute and verify the random points. We have discussed a possible solution, which is implemented in the current hybrid fuzzer, to handle two disjoint regions (i.e. one gap) through refining the constraints by rebasing them and compensate later.

We can make the computation efficient either by over-approximating to include gaps in between multiple disjoint ranges to pretend we have a single range, or by under-approximating to select one of the ranges to be only range. Note that the over-approximation does not guarantee the correctness anymore, whereas the under-approximation does. However, the first option may be more effective in increasing the code coverage when the ranges are scattered and large in size. With enough random inputs generated, losing *some* correctness doesn't hurt too much since the worst case in our setting is *random blackbox fuzzing*.

6.1.2 Improvements

The following improvements will not only advance the effectiveness of the hybrid fuzzer, but also boost the speed of it. These are the possible trivial extensions that can be added to the current system to provide better results.

Transformation rules. In `Input Generation` phase of the system, we implement the transformation rules for converting path predicates (which is in BIL) to PPL linear expressions. Some of these rules can be generalized in order to handle more general expressions. Also, the current implementation does not cover nested expressions, which can be extended by recursively transforming them into linear expressions. Non-linear formulas derived from the optimized code such as strlen or strcmp can be handled by heuristic rules, since they usually generate specific pattern of the bit operations. In fact, these routines can be transformed to naïve loop operations looking for a null byte or checking if the characters are the same.

Input source. When test cases are generated, we only handle either command line arguments or files that are read in by the target programs. Although these are the most common input sources that are fuzzed upon, we can easily support other input sources such as network sockets or `stdin` since Mayhem already support symbolic input introduction on these sources. Extending this feature enables fuzzing on network daemon services as well as interactive applications.

Pipelining. As mentioned earlier, all phases in the hybrid fuzzing system can be pipelined to yield better performance. This means that when the executors in `Symbolic Execution` phase terminate (due to hitting the end of the path), their path predicates can directly passed to `Input Generation` phase while other executors are still discovering paths. Then, when there are test cases generated, `Guided Fuzzing` phase can start feeding these test cases to monitor for abnormal behaviors. Current implementation is sequential in that each phase is blocked till the previous phase is finished, and thus hybrid fuzzing can be accelerated when the tasks are pipelined.

6.2 Future Work

In this section, we discuss the future works that can significantly improve our hybrid fuzzer and allow it to become more practical.

Dynamic Symbolic Execution Configuration. One of the reasons that our symbolic executor cannot reach all possible code paths in the target programs is due to the fixed configuration. By fixed configuration, we mean the configuration that defines symbolic input source and the length cannot be changed once the symbolic execution has started. This is a reasonable design choice to avoid any confusion and provide a deterministic path discovery. However, if the symbolic executor can automatically reason about the length of the symbolic bytes for certain input sources that is needed to exercise more code paths, we can significantly increase the breadth of the search.

Constraint Solver Integration. Although hybrid fuzzer can quickly create path-preserving random test cases, it is not able to handle non-linear predicates, which is critical since many paths (82%) contain non-linear predicates thus generated test cases do not guarantee the path execution. In order to resolve this issue, the hybrid fuzzer can invoke the constraint solver when it finds predicates that either it cannot handle or is non-linear. When we query the solver, we can also embed the current linear predicates in order to preserve the satisfiability when the random value is freely chosen from the possible input space described by these linear predicates. This way, we can precisely decide the values for the path critical symbolic bytes that are associated with non-linear constraints. This integration certainly introduces some overhead, but note that the hybrid fuzzer still significantly (75%) reduces the number of calls to the constraint solvers compared to traditional methods.

Adaptive Rejection Sampling. It is true that there can be a lot of wasted space in between the boundaries of the polytope and the hypercube, in case we have extreme distribution on the possible values for the input. By using *adaptive rejection sampling*, we can model the instrumental distribution using a set of piecewise exponential distributions rather than a single distribution and refine it as we perform the acceptance-rejection testing. This approach allows the hybrid fuzzer to keep reducing the space that will likely get rejected, cutting down the hypercube. This may reduce the number of rejections during the selection of the random points, but it comes with the cost. Point evaluation of log density is quite expensive, so in case there are not much gap between the hypercube and the polytope, it is usually more efficient to perform a normal rejection sampling.

Chapter 7

Related Work

In this chapter, we describe some previous works that are closely related to our research. We also compare and contrast our work with the related works to show possible improvements and extension to our system.

In the paper, *A Smart Fuzzer for x86 Exectuables* [29], Lanzi et al. introduce a fuzzing framework for x86 binary programs that improves the effectiveness of fuzzing compared to traditional random blackbox fuzzing. This smart fuzzer incorporates results from both static analysis and dynamic analysis to refine the input space for generating test cases. Instead of randomly choosing the values for the test case, the smart fuzzer tries to reason about the possible input space by statically analyze the binary, then monitors each execution to refine the input to lead the execution path to selected corner cases using constraint solver. However, this work only describes the hypothesis along with the possible design, but does not provide any implementation or evaluation of their work.

The above work is closely related to *Static Detection of Vulnerabilities in x86 Executables* [12], where the authors aim to identify security vulnerabilities in the binary programs by combining static analysis and symbolic execution. This approach is closer to traditional formal symbolic execution method, except they have added some heuristics (such as loop and recursion termination) to make the symbolic execution more practical against the real-world programs. Thus, the framework still requires to invoke constraint solvers to provide precise path-sensitive and context-sensitive analysis. Our hybrid fuzzer is different from this in that we aim to increase the code coverage while significantly reducing the number of calls to the constraint solvers, which essentially scales to the number of the discovered paths.

SAGE [21] is a hybrid fuzzer that has been developed and being continuously researched in Microsoft Research. In this work, Godefroid et al. provide a way to perform a *guided* execution by concolically executing the program and generate new inputs that will lead to different execution paths by negating and solving the path predicates to increase the code coverage. They describe a new search algorithm, called *generational search*, that aims to address some of the known limitations of traditional symbolic execution such as path explosion and imprecision. This system is similar in that the goal is to maximize the code coverage, but the order of the phases is the opposite of our hybrid fuzzing system. Specifically, the generational search algorithm first uses concrete inputs to dive (deep) into the code, then generates test cases by flipping the conditions from the end of the path predicate chain (which is repeated, possibly until all of the code has been covered). On the other hand, our hybrid fuzzer uses symbolic execution to spread in breadth first followed by random testing for the depth. Most critical distinction between two systems is the fact that our hybrid fuzzer does not require the solver for linear constraints, whereas SAGE needs to call the solver every time it needs to generate inputs for different paths.

Path-oriented random testing [23] is the closest related work of our research. The researchers of this work also realized that blind random testing does not perform well in case of complex constraints are present in the target program. In their work, they also try to deduce the over-approximation of the solutions of a set of constraints by performing constraint propagation. Our hybrid fuzzing system differs from this work in that we symbolically execute the program only in the beginning to spread out and transit into random testing from each frontier node such that we can achieve both the scalability (since random testing is fast) and the high code coverage (unique path guarantees along with random values). Also, in this related work, they simply ignore the *gap problem* where it can introduce false-positives in generating path-preserving test cases (i.e. it is prone to generate inputs that will not necessarily go down the path it is guaranteed by the system).

Finally, *TaintScope* [50] is a checksum-aware directed fuzzer that involves symbolic execution and taint analysis to automatically detect software vulnerabilities. While this work does not focus on increasing the code coverage itself, this can be a useful extension to the current system to be more practical fuzzer. In their research, Wang et al. aimed to solve one of the biggest limitations of traditional blackbox fuzzers: checksums. When the input is associated with checksum (somewhere in the input), most of the fuzzing runs will fail quickly without knowledge of the correct checksum due to the checksum verification routine in the target program. In this case, generating many test cases is far less effective since almost all of the test cases will not exercise more *interesting* part of the code. TaintScope addresses this issue by automatically identifying the verification routine, bypassing it when testing, and finally computes the correct checksum when needed using constraint solver.

Chapter 8

Conclusion

In our research, we have shown the benefits of hybrid fuzz testing technique in increasing the code coverage with relatively cheap overhead. Hybrid fuzzer provides an efficient way to generate provably random test cases that will guarantee the execution of the unique paths that they are generated from given that all predicates in a path predicate are linear. We also showed that the majority of the predicates is usually simple and linear.

We show that we can reduce the number of invocation to the constraint solvers at least 75% by solving linear predicates via polytope abstraction, and thus allowing more efficient test case generation. In case of some non-linear predicates, we adopt some heuristics to recognize common optimizations in order to convert these predicates to PPL linear expressions. We also discussed how non-linearity can be overcome by minimally invoking the constraint solver, which is still far less calls – due to such low number of non-linear predicates – compared to fully constraint solver dependent systems.

From the experiments we confirmed that random fuzzing is quite good at achieving high code coverage in the beginning of the testing but gets stuck soon after, whereas symbolic execution starts slowly but eventually discovers a lot more code than fuzzing. We also have shown that hybrid fuzzer lies on the *middle* as we hypothesized, allowing us to gain high code coverage in relatively short time. We only showed the results for fairly small programs, but as the size of the application becomes larger, we believe the hybrid fuzzing technique will be more effective: Fuzzing will not reason about complicated input constraints and the symbolic execution will take longer time to explore and query the constraint solvers for input generation.

Overall, the techniques we have proposed and developed in this thesis show that the efficient generation of effective inputs is possible. This can not only be used as a fuzzing system itself, but also extended to existing mechanisms to improve the performance. We learned that alternating the order of the phases such that random fuzzing is done first then followed by symbolic execution can be an interesting configuration possibility, since fuzzing is extremely fast at finding some initial code paths in the beginning and symbolic execution can leverage from each path that fuzzer already has found.

Appendix A

Formula Simplification

```
let change = ref false
let state_changed() = change := true
let unset_state() = change := false

let rec simplifications eval e =
   let xor_self = function
      | BinOp(XOR, e1, e2) when e1 === e2 ->
            state_changed();
            Int(bi0, get_type e1)
      | e -> e
   in

   let less_than_eq = function
      | BinOp(OR, BinOp(LT, e1, e2), BinOp(EQ, Int(i,_), BinOp(MINUS, e1', e2')))
      when e1 === e1' && e2 === e2' && i ==% bi0 ->
         state_changed();
         BinOp(LE, e1, e2)
      | BinOp(OR, BinOp(LT, e1, e2), BinOp(EQ, e1', e2'))
      when e1 === e1' && e2 === e2' ->
         state_changed();
         BinOp(LE, e1, e2)
      | BinOp(XOR,Cast(CAST_HIGH,Reg(1),BinOp(MINUS,e1,e2)),Cast(CAST_HIGH,Reg(1),BinOp(AND,BinOp(XOR,e3,e4),BinOp(XOR,
           e5,BinOp(MINUS,e6,e7)))))
      when e1 === e3 && e2 === e4 && e1 === e5 && e1 === e6 && e2 === e7 ->
         state_changed();
         BinOp(SLT, e1, e2)
      | BinOp(OR, BinOp(EQ, e1, e2), BinOp(XOR,Cast(CAST_HIGH,Reg(1),BinOp(MINUS,e3,e4)),Cast(CAST_HIGH,Reg(1),BinOp(
           AND,BinOp(XOR,e5,e6),BinOp(XOR,e7,BinOp(MINUS,e8,e9))))))
      when e1 === e3 && e2 === e4 && e1 === e5 && e2 === e6 && e1 === e7 && e1 === e8 && e2 === e9 ->
         state_changed();
         BinOp(SLE, e1, e2)
      | e -> e
   in

   let equal_to_val = function
      | BinOp(EQ, Int(i,_), BinOp(MINUS, e1, e2)) when i ==% bi0 ->
            state_changed();
            BinOp(EQ, e1, e2)
      | BinOp(AND, UnOp(NOT, BinOp(EQ, e0, e1)), BinOp(EQ, e2, e3)) when e1 === e2 ->
            state_changed();
            BinOp(EQ, e1, e3)
      | e -> e
   in

   let plus_zero = function
      | BinOp(PLUS, e1, Int(i,_)) when i ==% bi0 ->
            state_changed();
            e1
      | e -> e
   in

   let pad_zero = function
      | Concat(Int(i1,Reg(8)),Concat(Int(i2,Reg(8)),Concat(Int(i3,Reg(8)),e)))
      when i1 ==% bi0 && i2 ==% bi0 && i3 ==% bi0 ->
         state_changed();
         Cast(CAST_UNSIGNED, Reg(32), e);
   | e -> e
in

let is_zero = function
   | BinOp(EQ, Int(i1,_), Cast(CAST_LOW, Reg(8), Cast(CAST_UNSIGNED, Reg(32), UnOp(NOT, BinOp(EQ, e1, Int(i2,t2)))))
        )
   when i1 ==% bi0 && i2 ==% bi0 ->
      state_changed();
      BinOp(EQ, e1, Int(i2,t2))
   | e -> e
in

let and_true = function
   | BinOp(AND, e1, e2) when e1 === exp_true ->
         state_changed();
         e2
   | BinOp(AND, e1, e2) when e2 === exp_true ->
         state_changed();
         e1
   | e -> e
in
let vis = object(self)
   inherit Ast_visitor.nop
   method visit_exp e =
      let e =
         e
   |> xor_self
   |> less_than_eq
   |> equal_to_val
   |> plus_zero
   |> pad_zero
   |> is_zero
   |> and_true
in
'ChangeToAndDoChildren e
end
      in
      Ast_visitor.exp_accept vis e

let rec simplify eval e =
   unset_state();
   let e' = simplifications eval (eval e) in
   if(e' === e) then e' else simplify eval e'
```

Listing A.1: Formula Simplification Code in OCaml

Appendix B

strlen.c

```c
/* Copyright (C) 1991, 1993, 1997, 2000, 2003 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
   Written by Torbjorn Granlund (tege@sics.se),
   with help from Dan Sahlin (dan@sics.se);
   commentary by Jim Blandy (jimb@ai.mit.edu).

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

#include <string.h>
#include <stdlib.h>

#undef strlen

/* Return the length of the null-terminated string STR.  Scan for
   the null terminator quickly by testing four bytes at a time.  */
size_t
strlen (str)
     const char *str;
{
  const char *char_ptr;
  const unsigned long int *longword_ptr;
  unsigned long int longword, magic_bits, himagic, lomagic;

  /* Handle the first few characters by reading one character at a time.
     Do this until CHAR_PTR is aligned on a longword boundary.  */
  for (char_ptr = str; ((unsigned long int) char_ptr
                        & (sizeof (longword) - 1)) != 0;
       ++char_ptr)
    if (*char_ptr == '\0')
      return char_ptr - str;

  /* All these elucidatory comments refer to 4-byte longwords,
     but the theory applies equally well to 8-byte longwords.  */

  longword_ptr = (unsigned long int *) char_ptr;

  /* Bits 31, 24, 16, and 8 of this number are zero.  Call these bits
     the "holes."  Note that there is a hole just to the left of
     each byte, with an extra at the end:

     bits:  01111110 11111110 11111110 11111111
     bytes: AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD

     The 1-bits make sure that carries propagate to the next 0-bit.
     The 0-bits provide holes for carries to fall into.  */
  magic_bits = 0x7efefeffL;
  himagic = 0x80808080L;
  lomagic = 0x01010101L;
  if (sizeof (longword) > 4)
    {
      /* 64-bit version of the magic.  */
      /* Do the shift in two steps to avoid a warning if long has 32 bits.  */
      magic_bits = ((0x7efefefeL << 16) << 16) | 0xfefefeffL;
      himagic = ((himagic << 16) << 16) | himagic;
      lomagic = ((lomagic << 16) << 16) | lomagic;
    }
  if (sizeof (longword) > 8)
    abort ();

  /* Instead of the traditional loop which tests each character,
     we will test a longword at a time.  The tricky part is testing
     if *any of the four* bytes in the longword in question are zero.  */
  for (;;)
    {
      /* We tentatively exit the loop if adding MAGIC_BITS to
         LONGWORD fails to change any of the hole bits of LONGWORD.

         1) Is this safe?  Will it catch all the zero bytes?
         Suppose there is a byte with all zeros.  Any carry bits
         propagating from its left will fall into the hole at its
         least significant bit and stop.  Since there will be no
         carry from its most significant bit, the LSB of the
         byte to the left will be unchanged, and the zero will be
         detected.

         2) Is this worthwhile?  Will it ignore everything except
         zero bytes?  Suppose every byte of LONGWORD has a bit set
         somewhere.  There will be a carry into bit 8.  If bit 8
         is set, this will carry into bit 16.  If bit 8 is clear,
         one of bits 9-15 must be set, so there will be a carry
         into bit 16.  Similarly, there will be a carry into bit
         24.  If one of bits 24-30 is set, there will be a carry
         into bit 31, so all of the hole bits will be changed.

         The one misfire occurs when bits 24-30 are clear and bit
         31 is set; in this case, the hole at bit 31 is not
         changed.  If we had access to the processor carry flag,
         we could close this loophole by putting the fourth hole
         at bit 32!

         So it ignores everything except 128's, when they're aligned
         properly.  */

      longword = *longword_ptr++;

      if (
#if 0
          /* Add MAGIC_BITS to LONGWORD.  */
          (((longword + magic_bits)

            /* Set those bits that were unchanged by the addition.  */
            ^ ~longword)

           /* Look at only the hole bits.  If any of the hole bits
              are unchanged, most likely one of the bytes was a
              zero.  */
           & ~magic_bits)
#else
          ((longword - lomagic) & himagic)
#endif
          != 0)
        {
          /* Which of the bytes was the zero?  If none of them were, it was
             a misfire; continue the search.  */

          const char *cp = (const char *) (longword_ptr - 1);

          if (cp[0] == 0)
            return cp - str;
          if (cp[1] == 0)
            return cp - str + 1;
          if (cp[2] == 0)
            return cp - str + 2;
          if (cp[3] == 0)
            return cp - str + 3;
          if (sizeof (longword) > 4)
            {
              if (cp[4] == 0)
                return cp - str + 4;
              if (cp[5] == 0)
                return cp - str + 5;
              if (cp[6] == 0)
                return cp - str + 6;
              if (cp[7] == 0)
                return cp - str + 7;
            }
        }
    }
}
libc_hidden_builtin_def (strlen);
```

Listing B.1: `strlen` implementation in glibc

Appendix C

Synthetic Programs for Experiments

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
   unsigned char *input = argv[1]; // unsigned byte
   char *input2 = argv[2];         // signed byte

   if(input[2] >= 20)
   {
      if(input2[0] <= -80)
      {
         printf("PATH 1\n");
         return 0;
      }
      printf("PATH 2\n");
      return 0;
   }
   printf("PATH 3\n");

   return 0;
}
```

Listing C.1: `simple.c` code

```c
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
   int fd;
   char buf[8];
   size_t count;

   fd = open("readme", O_RDONLY);
   if(fd == -1) {
      perror("open");
      exit(-1);
   }

   count = read(fd, buf, 8);
   if(count == -1) {
      perror("read");
      exit(-1);
   }

   if(buf[0] != 'b')
   {
      if(buf[2] < 'h')
      {
         printf("BUG!\n");
         close(fd);
         return 0;
```

```c
        }
        printf("BUG2!\n");
        close(fd);
        return 0;
    }
    printf("BUG3!\n");
    close(fd);

    return 0;
}
```

Listing C.2: `simple_file.c` code

```c
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char** argv)
{
    int fd = open("readme", O_RDONLY);
    int input;
    read(fd, &input, sizeof(input));
    if(input >= -0x41414141)
    {
      if(input < -0x10101010)
      {
         printf("BUG 1\n");
         return 0;
      }
       printf("BUG 2\n");
       return 0;
    }
    printf("BUG 3\n");
    return 0;
}
```

Listing C.3: `int_cmp.c` code

Bibliography

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, February 2011. 2

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. URL `http://bugseng.com/products/ppl/documentation/BagnaraHZ06TR.pdf`. 4.2

[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, June 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.08.001. URL `http://dx.doi.org/10.1016/j.scico.2007.08.001`. 3.2

[4] blexim. Basic integer overflows. (January 2003), 2003. URL `http://www.phrack.org/archives/60/p60_0x0a_Basic%20Integer%20Overflows_by_blexim.txt`. 2.1.1

[5] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, MIT, PhD Thesis, 2004. 3.2

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL `http://dl.acm.org/citation.cfm?id=2032305.2032342`. 2.1.2

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM*

conference on Computer and communications security, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. doi: 10.1145/1180405.1180445. URL http://doi.acm.org/10.1145/1180405.1180445. 1.1, 2, 2.2.2

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855756. 2.2.2

[9] Kun chan Lan, Alefiya Hussain, and Debojyoti Dutta. Effect of malicious traffic on the network, 2003. 1.2

[10] Hong-Zu Chou, I-Hui Lin, Ching-Sung Yang, Kai-Hui Chang, and Sy-Yen Kuo. Enhancing bug hunting using high-level symbolic simulation. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, GLSVLSI '09, pages 417–420, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-522-2. doi: 10.1145/1531542.1531637. URL http://doi.acm.org/10.1145/1531542.1531637. 1.1

[11] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM. doi: 10.1145/512760.512770. URL http://doi.acm.org/10.1145/512760.512770. 3.1

[12] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 269–278, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2716-7. doi: 10.1109/ACSAC.2006.50. URL http://dx.doi.org/10.1109/ACSAC.2006.50. 2.1.2, 7

[13] Dewey and Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, 5th February - 8th February 2011*. The Internet Society, 2012. 2.1.2

[14] M. Eddington. Peach fuzzing platform. (June 2007), 2007. URL http://peachfuzzer.com/. 2.2.1

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512558. URL `http://doi.acm.org/10.1145/512529.512558`. 2

[16] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070546. URL `http://dx.doi.org/10.1109/ICSE.2009.5070546`. 2.2.1

[17] GNU GCC. Invoking gcov, 2010. URL `http://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html`. 3.1

[18] Patrice Godefroid. 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, pages 1–1, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-881-7. doi: 10.1145/1292414.1292416. URL `http://doi.acm.org/10.1145/1292414.1292416`. 2.2.1

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065036. URL `http://doi.acm.org/10.1145/1065010.1065036`. 2, 2.2.2

[20] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375607. URL `http://doi.acm.org/10.1145/1375581.1375607`. 1.1, 2.2.1

[21] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008. 7

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012. ISSN 1542-7730. doi: 10.

1145/2090147.2094081. URL http://doi.acm.org/10.1145/2090147.2094081. 1.1, 2.2.1, 2.2.2

[23] Arnaud Gotlieb and Matthieu Petit. Path-oriented random testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 28–35, New York, NY, USA, 2006. ACM. ISBN 1-59593-457-X. doi: 10.1145/1145735.1145740. URL http://doi.acm.org/10.1145/1145735.1145740. 7

[24] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6. 1.1

[25] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: 10.1145/1028664.1028717. URL http://doi.acm.org/10.1145/1028664.1028717. 2

[26] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252. 2.2.2

[27] Caca Labs. zzuf multi-purpose fuzzer. (January 2010), 2010. URL http://caca.zoy.org/wiki/zzuf. 2.2.1, 3.2, 5.3

[28] Caca Labs. List of bugs found by zzuf. (January 2010), 2010. URL http://caca.zoy.org/wiki/zzuf/bugs. 5.3

[29] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2952-6. doi: 10.1109/SESS.2007.1. URL http://dx.doi.org/10.1109/SESS.2007.1. 7

[30] Codenomicon Ltd. Defensics fuzzer. (January 2001), 2001. URL http://www.codenomicon.com/defensics/. 2.2.1

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN

1-59593-056-6. doi: 10.1145/1065010.1065034. URL http://doi.acm.org/10.1145/1065010.1065034. 1.1, 3.2

[32] Tilo M. Aslr smack & laugh reference seminar on advanced exploitation techniques. (June 2005):1–21, 2008. URL http://netsec.cs.northwestern.edu/media/readings/defeating_aslr.pdf. 2.1.1

[33] C. Miller. How smart is intelligent fuzzing - or - how stupid is dumb fuzzing? (June 2007), 2007. URL https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-miller.pdf. 2.2.1

[34] Jelena Mirkovic, Sonia Fahmy, Peter Reiher, Roshan Thomas, Alefiya Hussain, Steven Schwab, and Calvin Ko. P.: Measuring impact of dos attacks. In *In: Proceedings of the DETER Community Workshop on Cyber Security Experimentation. (2006)*. 1.2

[35] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *ACSAC*, pages 421–430, 2007. 2.1.2

[36] Quinn Norton. Antisec hits private intel firm; millions of docs allegedly lifted, 2011. URL http://www.wired.com/threatlevel/2011/12/antisec-hits-private-intel-firm-million-of-docs-allegedly-lifted. 1.2

[37] T. Ormandy. fuzz. (June 2007), 2007. URL http://freecode.com/projects/taviso-fuzz. 2.2.1

[38] Hass P. Advanced format string attacks. (June 2010), 2010. URL https://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf. 2.1.1

[39] Daniel Quinlan and Thomas Panas. Source code and binary analysis of software defects. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 40:1–40:4, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-518-5. doi: 10.1145/1558607.1558653. URL http://doi.acm.org/10.1145/1558607.1558653. 1.1

[40] Alexandre Rebert Sang Kil Cha, Thanassis Avgerinos and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, May 2012. 2.2.2, 5.3

[41] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081750. URL `http://doi.acm.org/10.1145/1081706.1081750`. 2, 2.2.2

[42] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE '08, pages 299–304, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68854-9. doi: 10.1007/978-3-540-68855-6_19. URL `http://dx.doi.org/10.1007/978-3-540-68855-6_19`. 2.2.1

[43] B. Sineath. Static binary analysis of recent smbv2 vulnerability. (October 2009), 2009. URL `http://www.secureworks.com/research/threats/windows-0day/`. 2.1.2

[44] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008. 2.1.2

[45] Alexander Sotirov. Automatic vulnerability detection using static source code analysis. Technical report, 2005. 2.1.1

[46] I. Sprundel. Fuzzing: Breaking software in an automated fashion. (December 2005), 2005. URL `http://events.ccc.de/congress/2005/fahrplan/attachments/582-paper_fuzzing.pdf`. 2.2.1

[47] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN 0321446119. 4.1

[48] A. Takanen. Fuzzing: the past, the present and the future. (June 2009), 2009. URL `http://actes.sstic.org/SSTIC09/Fuzzing-the_Past-the_Present_and_the_Future/SSTIC09-article-A-Takanen-Fuzzing-the_Past-the_Present_and_the_Future.pdf`. 2.2.1

[49] L.A. Times. Sony pictures says lulzsec hacked 37,500 user accounts, not 1 million, 2011. URL `http://www.webcitation.org/5zNXkePWU`. 1.2

[50] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Trans. Inf. Syst. Secur.*, 14 (2):15:1–15:28, September 2011. ISSN 1094-9224. doi: 10.1145/2019599.2019600. URL `http://doi.acm.org/10.1145/2019599.2019600`. 2.2.1, 7