CHALMERS

Multi-Platform Binary Program Testing Using Concolic Execution

Master's Thesis in Computer Systems and Networks

EIKE SIEWERTSEN

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden 2015

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Multi-Platform Binary Program Testing Using Concolic Execution

© Eike Siewertsen, September 2015.

Examiner: Magnus Almgren
Supervisor: Vincenzo Massimiliano Gulisano

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2015

Abstract

In today's world, computer systems have long since made the move from the desktop to numerous locations in the form of routers, intelligent light bulbs, electricity meters and more. These devices are referred to as "embedded devices", because they are generally smaller, less powerful versions of their PC counterparts. These computers have lower memory and CPU performance requirements, leading to challenges in creating software for such devices: resource constraints encourage the use of unsafe programming languages, which in turn promotes an environment that can lead to mistakes or unintended side-effects of the created program. Some of these errors open up vulnerabilities that can be abused, causing the program to perform unintended actions or giving the actors control over the software and consequently the device. For the owner of the device, this can lead to annoyances in the case of an intelligent light bulb, to real, economic and physical damage in the case of devices in a critical infrastructure.

It is therefore necessary for a manufacturer of such devices to ensure the presence of steps in the development process that ensure the software conforms rigorously to the specification. Aside from development guidelines for writing program code, and the choice of safe programming languages, the actual testing of the finished application can discover critical errors before the release of the program or device. Testing applications can be time-intensive work due to the need for proper test cases that elicit the entirety of the program's functionality. For this reason, a suite of applications exists called "fuzzers", that automatically generate exhaustive test cases for programs.

In this thesis we present an automated system for generating such test cases for a program. Unlike most fuzzing engines, our solution can function without the need for its source code, and it supports multiple architectures. We use a technique called "concolic execution" to ensure that each newly generated test case causes the program to exhibit new behavior. With this, we intend to be able to cover all parts of the original program much faster than humans could manually write test cases and better than completely randomly generated inputs.

Our results are promising in that they show that we can produce exhaustive test cases that reliably cause a program to follow unique execution paths. This accuracy comes with a greatly increased execution cost, however, causing our current solution to require further work to handle larger programs. Nevertheless, we can show that in isolation, our solution outperforms an advanced fuzzer in the case of complicated branch conditions. We conclude that the area of our thesis shows promising results, and that future work can greatly improve the result by removing some of the current limitations and improving performance.

Acknowledgements

Thanks are extended primarily to my thesis advisors Magnus Almgren and Vincenzo Massimiliano Gulisano for the time they took to guide me through the creation of this work, their great advice on writing and presentation techniques and their time taken to provide feedback. Further thanks are extended to the PANDA community and their developers. Particularly Patrick Hulin in his work on the taint plugin, and advice on modifying it. I would also like to thank Victor Lopez and Daniel Schoepe for proofreading and providing a much needed second opinion.

Eike Siewertsen, Gothenburg 2015

Contents

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Contributions 3
- 1.3 Organization 4
- 1.4 Ethics and Sustainability 4

2 Background **5**
- 2.1 Dynamic Taint Analysis 5
  - 2.1.1 Shadow Memory 6
- 2.2 Symbolic Execution 6
  - 2.2.1 Concolic Execution 7
- 2.3 LLVM 7
- 2.4 PANDA 8
  - 2.4.1 QEMU 8
  - 2.4.2 LLVM JIT 9
  - 2.4.3 Record & Replay 9
- 2.5 Satisfiability Modulo Theories 10

3 Related Work **13**
- 3.1 Fuzzing Engines 13
- 3.2 Symbolic Execution 14
  - 3.2.1 Concolic Execution 15
  - 3.2.2 Hybrid Execution 16
- 3.3 Summary 17

4 Design and Implementation **19**
- 4.1 System Structure 19
- 4.2 Input Files 22

4.2.1 Scoring . . . 22
4.3 Recording . . . 23
4.4 Branch Constraints . . . 23
4.4.1 LLVM Instrumentation . . . 24
4.4.2 Memory I/O . . . 25
4.4.3 Operator Translation . . . 27
4.4.4 Symbolic Pointers . . . 28
4.5 Test Case Generation . . . 28

5 Evaluation **31**
5.1 Methodology . . . 31
5.2 Tests . . . 33
5.2.1 Complex Branch Conditions . . . 33
5.2.2 Complex Control Flow Logic . . . 34

6 Future Work **39**
6.1 Error Detection . . . 39
6.2 Performance Improvements . . . 40
6.3 Improved Input Scoring . . . 41

7 Conclusion **43**

Bibliography **49**

List of Figures

1.1 CVEs Published per Year [CVE] 2

2.1 The QEMU to TCG to LLVM to Host Instructions Translation Pipeline [CC10] 10
2.2 SMT-LIB Example of the Bitvector and Array Theory 11

4.1 Flowchart of the project's structure 20
4.2 An example of concolic execution with branch constraints in our design 23
4.3 Example of two memory reads, showing symbolic and concrete memory 26
4.4 Example of a symbolic write to memory 27
4.5 Test case generation 29

5.1 `crackaddr` test results 35
5.2 `mime1` test results 36
5.3 `prescan` test results 36

List of Tables

4.1 Different types of shadow memory 25
4.2 Result of extracting bytes from the expression in Figure 4.4 27

5.1 Results of the performed tests 34
5.2 Execution time per generated test case 37

Glossary

AFL American Fuzzy Lop - A high-performance greybox fuzzer

Basic Block A sequence of assembly instructions with a single entry and exit point. It either ends in a control flow statement (such as a jump instruction), or the instruction following the last instruction is the target of another control flow statement

IR Intermediary Representation - An internal representation of program code, generally used by compilers before generating architecture-specific assembly code

LLVM An extensive, open-source compiler tool chain, including a well-documented intermediary language and corresponding API

PANDA A project built upon QEMU that can record and replay machine executions

QEMU An open source machine emulator

TCG Tiny Code Generator - QEMU's module to translate assembly between architectures

Z3 An open-source SMT solver developed by Microsoft

1

Introduction

The relatively young craft of software development has its share of unsolved problems. From mismanagement of developer teams and bad planning of large-scale software systems, to the difficulties from some computational impossibilities in computer science [Tur36], the field still remains in constant motion. One of the problems is the security of the system by ensuring confidentiality, integrity and availability of data or services. It is generally accepted that no computer program can achieve full security[1], largely because the three properties influence and cancel each other out. However, achieving as much as possible of this goal still remains a primary focus in computer security, because of the damage (financial, privacy and otherwise) involved in security breaches.

That the number of vulnerabilities is not decreasing, is shown by the data collected by the National Vulnerability Database (NVD) [CVE]. This is a database of all publicly reported vulnerabilities in computer systems, run by the U.S. government. Each vulnerability is assigned a Common Vulnerabilities and Exposures (CVE) ID and registered in the NVD and other systems. Figure 1.1 lists the number of CVEs published per year, and shows that computer system security remains a problematic area.

The importance of secure computing systems is even more reinforced by the ubiquity of computing devices in today's society: The "Internet of Things" (IoT) describes common household objects outfitted with computer chips (embedded computers) that are connected to each other. This ranges from cell phones, cameras and radios to intelligent fridges and electricity meters. These devices often operate on very low memory and processing power, requiring developers to operate under strong constraints. Compromise of such devices can lead to economical damage,

[1] http://c2.com/cgi/wiki?ComputerSecurityIsImpossible

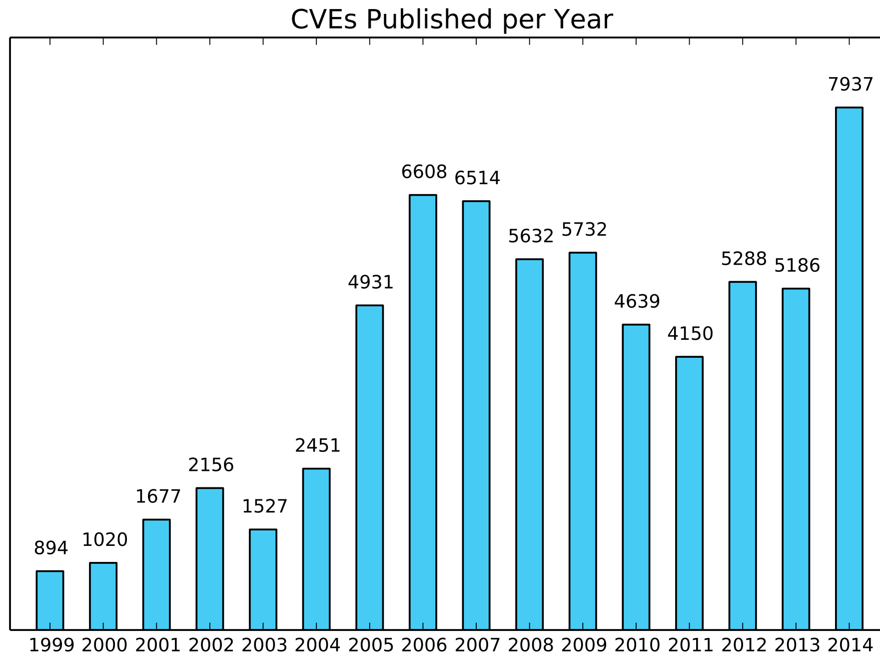


Figure 1.1: The number of CVEs published in the National Vulnerability Database per year (data from [CVE])

or even personal damage (in the case of pacemakers), depending on their field of operation.

Proper auditing of computer systems before deployment, along with strict programming guidelines, the usage of safe programming language or even formal verification can limit the chance of problematic vulnerabilities that will later be exploited by attackers. However, most of these methods remain very time-intensive, and thus, expensive, preventing their wide-spread adoption. In the case of embedded computer systems, there simply might not be the resources to keep certain security measures in place.

1.1 Motivation

Progress has been made in the development of automated test-case generation tools, also called fuzzers[2], that continuously run the target program with new,

[2] `http://pages.cs.wisc.edu/~bart/fuzz/`

randomized, inputs, in order to cause it to enter an unexpected state, crash, or exhibit other misbehavior that could potentially be exploited. A limitation of these randomized fuzzing tools is, that, since they do not posses any knowledge about the input format of the tested program, they rarely generate well-formed inputs. This leads to the execution never progressing very far into the program, and only testing shallow execution paths. Attempts have been made at more intelligent test-case generation through grammars [GKL08] or parsing of source code [GKS05], but are still connected with time-intensive tasks to set up the fuzzers.

A measure of the effectiveness of fuzzers, is the lines of code covered during one execution caused by a test case. As such, the primary goal of fuzzers and our work is, to cover as many unique lines of code with as few test cases as possible. The act of working towards this goal is called code coverage maximization.

Research has been conducted in developing more intelligent fuzzing frameworks using binary analysis, as described in more detail in Chapter 3, however most of it is focused on desktop computer systems or its results are simply not made available publicly. Therefore this thesis's focus is on bridging the gap: Development of an intelligent fuzzing method using the binary analysis method symbolic execution, described in Chapter 2, that remains as platform-agnostic as possible and can work with binary programs without the need for source code.

1.2 Contributions

The contributions of this thesis are primarily in the form of a design and implementation of symbolic execution on top of a framework that supports a number of beneficial properties. Unlike most fuzzing engines, our solution, except for evaluation purposes, works completely without knowledge of the program source code. This presents two primary benefits: First it gives users of proprietary software a method to test it for errors, second, by operating on the compiled binary, it makes it possible to catch errors that only manifest after compilation. Additionally, since the solution is based on the QEMU emulator, support for multiple chipsets, including X86, X64 and ARM is possible. With the advent of widespread embedded devices this is particularly appealing.

As such, our contributions are:

- A platform to dynamically generate test files for programs that cause the program to behave differently.
- Analysis of binary programs with zero knowledge about the program internals, without any source code, and on multiple architectures.
- For cases where the source code is available, a method to measure source line coverage.

- A method to rank generated test files by their likeliness to cover new code.

1.3 Organization

In Chapter 2 we give an in-depth introduction into the theory needed to understand the area of this work and its implementation. Chapter 2 continues with an overview over similar research in our field, describing the different approaches to automated program testing and ultimately the reason why our approach is novel. We go on to describe the details of our implementation in Chapter 4, so that we can then elaborate on the evaluation of it in Chapter 5. Finally, we discuss the results and their implications in Chapter 6, along with potential future areas of work.

1.4 Ethics and Sustainability

In this work, we develop a system that is intended to be used to discover security vulnerabilities in programs. It is intended to be used by researchers or developers during development of security-critical applications. As with every security research software, it can also be used by malicious actors to discover and exploit vulnerabilities. This faces us with two choices: We can either withhold or release this work. While not releasing it would intuitively prevent any abuse, it does not prevent the existence of works similar to us. In fact, a number of similar open-source and closed-source solutions already exist. By releasing our work however, we can ensure that security researchers and developers benefit from our work, and are able to improve the security of applications.

Aside from electricity, our work does not directly use any depletable resources. The source of the electricity can be picked to come from renewable sources.

2

Background

This section will give a broad introduction into the concepts, algorithms, and data formats this thesis depends on. We will describe the techniques necessary for symbolic execution and the various frameworks that have been used in this thesis's work.

2.1 Dynamic Taint Analysis

In the area of program analysis, one of the main goals is to discover what transformations the program applies to a piece of data [SAB10]. This initial piece of data, the taint source, needs to be defined at the start, and usually consists of some form of input to the program, for example, files, command line arguments or network traffic. Dynamic taint analysis then tracks the flow of data through the program and identifies the operations performed on it. The result of these operations is then also considered tainted, while any other data is considered untainted. In contrast to static taint analysis, which tracks information flow without execution of a program, dynamic taint analysis does so by executing a program augmented with callbacks, which notify the algorithm of the state of execution.

This fundamental technique opens up a number of uses in program analysis [SAB10], where the particulars depend on the concrete use case. In malware analysis, for example, one might consider sensitive information on the computer (private documents, passwords, etc.) tainted and be alerted if any tainted data ever reaches network functions, which might suggest unwanted exfiltration of data. In program testing, the interest would largely be on what instructions operate on the input data, so it could be used to generate better input data, leading to a higher code coverage.

Concretely, the tracking of taint through operations requires a good understanding of the semantics. Misrepresentations can lead to two potential errors: When a derived value is marked as tainted, while, in reality, it is not, it is considered *overtainted*. Missing a flow of taint from an operation, and thus not marking an actually tainted value as tainted is called *undertainting*. In some cases it might be impossible to correctly propagate taint through an operation, due, for example, uncertain semantics of intrinsic functions (I/O operations, hardware instructions) or other non-deterministic operations. For these cases, the decision whether to overtaint or undertaint depends on the particular use case. In malware analysis it might, for example, be better to produce a false positive than a false negative.

2.1.1 Shadow Memory

Taint analysis requires bookkeeping of the taint state of the systems memory, including I/O memory and CPU registers, in order to properly propagate taint. This is sometimes called shadow memory [WLK13], the size of which depends on the granularity of the analysis. When accuracy (i.e. as little overtainting as possible) is desired, the shadow memory can track the taint state for every individual byte. On the other hand, if accuracy is less important, and a certain amount of overtainting is an acceptable compromise, it could be lifted to word (two bytes) level. Depending on the accuracy, each unit of memory needs to at least track the taint state, i.e. one bit for whether the data is tainted or not. In reality though, more information about where the taint is coming from is usually desired, leading to a larger amount of information being tracked with each unit of memory.

Depending on the number of expected instructions, this leads to the need for an efficient implementation of the shadow memory. A compromise needs to be made between execution speed (lookup speed of the taint state) and memory usage, since at taint state sizes of at least four bytes, it can quickly become impossible to fit the fully allocated shadow memory in the operating systems memory, and perform dynamic analysis.

2.2 Symbolic Execution

Symbolic execution is a way of extending taint analysis in order to learn about how exactly the inputs to the program affect its execution [SAB10, Kin76, How77]. While taint analysis simply tracks *which* inputs reached the output, symbolic execution also tracks *how* the input has been modified. For this, the program is executed while all its inputs are treated as symbolic variables. Any transformations that happen on these inputs during the execution are tracked in the form of logical formulas depending on these symbolic variables. For the system performing

the execution, these symbolic values are kept in a symbolic state that mirrors the memory locations of the system: Register, RAM and even hard disk storage.

As such, in addition to the mentioned symbolic state, symbolic execution maintains a path constraint (`PC`) in the form of a list of logical first-order expressions depending on the input variables. Each time the execution arrives at a conditional branch, the process forks. In one fork, the logical expression of the conditional is added to `PC`, in another the negation of the expression is added. Through this, both branches of the branch are covered. However, due to the existance of loops which could be infinite, this forking process has to be restricted in some manner.

The forking process is one of symbolic execution's great weaknesses: The exponential path explosion that follows each branch, leads to most naive implementations of it not scaling well to larger applications. Since every fork's symbolic state is kept in parallel, the system's resources will eventually run out. Therefore, most systems implement some form of path selection algorithm that prevents uninteresting directions of branches to be explored further.

2.2.1 Concolic Execution

Concrete **o**nline symbo**lic** (concolic) execution is a more restricted variant of symbolic execution [QR11]. This variant restricts itself to analyzing a single execution of a program, i.e. it does not fork on branches. The result of this is, that instead of covering all possible execution paths in one execution, concolic execution outputs the **PC** that is recorded during a single execution. A resulting **PC** can then be used to generate new input files by selectively negating individual branch constraints and solving the resulting, new, path constraint.

The advantages of this approach are, that concolic execution for program testing generally scales better than pure symbolic execution. This is because only a single program trace is ever analyzed at a time. The problem of infinite loops does not disappear, however, but as long as an algorithm exists to prune uninteresting execution paths, concolic execution can iteratively analyze each path without exhausting resources due to the path explosion problem.

2.3 LLVM

LLVM is a compiler infrastructure project started as a research project in the University of Illinois [Lat02]. It has since turned into a large open-source project. Its core is an intermediary language of the same name ("LLVM Intermediate Representation", or "LLVM IR". When we refer to LLVM in the following sections, we mean the IR) and a supporting optimizer for it. It supports native code generation from its IR into a large number of architectures. It provides a well-developed API

for its IR and consists of a small number of well-defined instructions. Extended information can be read in its language reference manual[1].

2.4 PANDA

The PANDA framework [DGHH14] has been selected to be used as a base for the symbolic execution technique implemented in this thesis. While relatively young and under constant development, PANDA's unique record & replay structure, coupled with an LLVM translation unit extracted from the S^2E project [CKC11] makes it an ideal framework for this thesis's work. In this section, we elaborate on the parts of PANDA necessary to understand the rest of this thesis.

PANDA was first specifically described in the technical report "Repeatable Reverse Engineering for the Greater Good with PANDA" by Dolan-Gavitt et al. [DGHH14]. It is built upon the QEMU whole-system emulator [Bel05] and extends it with the ability to record system executions, store them on disk and later replay them in exactly the same way they previously executed.

To extend analysis techniques performed on the program during execution, PANDA uses a plugin architecture, allowing for individual plugins to be enabled and disabled during runtime and to define APIs to interact between plugins.

As mentioned, PANDA is based on the QEMU[2] [Bel05] open source emulator. It allows operating systems, and programs running inside it, of one instruction set to be executed on a host system of another instruction set. At the time of this writing, it primarily supports x86, x86-64, ARM, MIPS, PowerPC and other chipsets. Aside from its wide support of architectures, its main features are its speed together with its relative simplicity.

2.4.1 QEMU

Guest code translation is done through a Just in Time (JIT) dynamic binary translator: Individual guest basic blocks are converted to a simple intermediary language (IL) by the Tiny Code Generator (TCG) and then translated to the host system instruction set. The result of each translated basic block is cached, so that subsequent invocations of them are performed faster.

While the TCG IL makes it easy for a developer to implement a new compiler frontend by only having to work with a single language, it is not suitable for analysis, for the same reason that assembly languages are not: It is not side-effect free.

[1] http://llvm.org/docs/LangRef.html
[2] http://wiki.qemu.org/

Side-effects of an assembly language are commonly present in form of secondary non-obvious operations performed by a specific instruction. For example, in the x86 assembler, a single `add` instruction is expected to store the sum of two registers in another destination register. It does however also modify a number of bit-sized registers called the "condition codes". Here, the CPU stores information on whether the previous instruction has lead to an overflow, whether the result was zero or negative, and others. Condition branch instruction that jump to other locations in the code then depend on these condition codes. These side-effects do however make binary analysis on pure assembly difficult, since they have to be properly modeled for every single instruction [FPS11, SBY+08].

TCG shares these properties, in that it delays calculation of condition codes until they are actually required, when execution reaches a branch instruction.

2.4.2 LLVM JIT

Since assembly languages and the TCG IL have side-effects, PANDA integrates a module [CC10, CGZC09] from the S^2E project that implements an LLVM code backend for the TCG. The module translates each TCG operation to an equivalent number of LLVM instructions, and constructs entire LLVM basic blocks by chaining together several calls to these instructions into one LLVM function per basic block. Afterwards, the LLVM code optimization passes are applied to each translated basic block, as the originally generated code is far from optimal, due to unused variables and unnecessary indirections due to the number of calls. The generated LLVM code can then be translated back to the host system's instruction set, using LLVM's built in JIT-compiler. Figure 2.1 details the translation process inside QEMU, leading to LLVM-generated assembly. At the top, X86 guest code is translated to the internal TCG micro-operations, at which point the TCG OP to host instruction mapping is used to access equivalent LLVM snippets, which are then combined into the equivalent LLVM bitcode.

As detailed in [CC10] however, the translation process is significantly more expensive than the original TCG translation process. This is due to the fact that for each basic block the full LLVM optimizer and compiler has to be run. The higher emulation slowdown leads to the original work in [CC10] having issues with timeouts in operating systems with LLVM translation enabled. PANDA solves this problem with its record & replay functionality, described in the next section.

2.4.3 Record & Replay

For this, during execution, PANDA writes all inputs to the CPU, hardware interrupts and RAM read/writes to a "non-determinism" log, along with the time when the event occurred. Recording can be started at any point during QEMU's

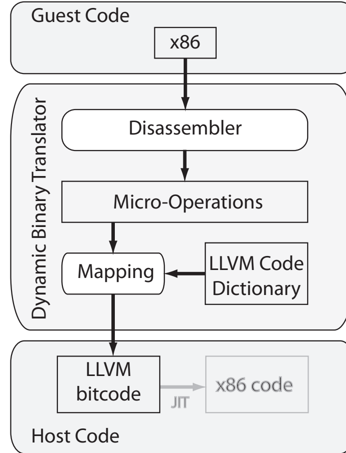


Figure 2.1: The QEMU to TCG to LLVM to Host Instructions Translation Pipeline [CC10]

execution. If recording begins after the guest system is already running, a QEMU snapshot is first created, which captures the entire system's state and can be loaded again at any point. This information is then enough to accurately replay the operations executed inside QEMU, and repeat the recorded process.

At this point the previously mentioned LLVM translation becomes viable: Since the program's execution no longer depends on any outside inputs such as timing information, the overhead imposed by LLVM translation cannot cause timeouts inside the guest operating system or application anymore. Instead, PANDA analysis plugins, such as the one responsible for taint flow analysis, can take enough time to properly perform their work.

2.5 Satisfiability Modulo Theories

In computer science, the boolean satisfiability (SAT) problem deals with determining if an interpretation of boolean variables for a given boolean formula exists [Coo71]. Since this problem is generally NP-complete, there exist no algorithms that can confirm or deny the existence of such an interpretation with a non-exponential worse-case running time. Instead, however, there exist a number of *efficient* solvers that, while still having an exponential worst-case running time, can solve the problem very efficiently for most formulas that arise in practice [DP60, DLL62]. Satisfiability modulo theories (SMT) [BSST09] builds upon SAT by extending it with a number of theories from first-order logic, such as reasoning about equality, arithmetic, natural/real numbers, bit-vectors and arrays. Similarly, a number of fast solvers exist for the SMT problem [Tin02].

One such solver is Z3 [DMB08], developed at Microsoft by Leonardo de Moura and Nikolaj Bjørner. Z3 differentiates itself from other solvers by providing well-developed APIs for C, C++ and Python to a very fast open source SMT solver.

A standardized format of representing SMT formulas is that of SMT-LIB[3]. The Lisp-like language documents syntax and semantics for representing the different theories, access to the solver and solver extensions.

For this thesis, a number of theories are relevant. The theory of fixed-size bitvectors deals with logical and arithmetic operations and extraction and concatenation on bitvectors. The arithmetic (addition, subtraction, etc.) and logical (and, or, etc.) operations represent those happening in the CPU, and thus include side-effects such as overflows or underflows. Arithmetic instructions are generally repeated in two versions: one where the operation treats its arguments as unsigned, one as signed. Examples for arithmetic operations are `bvadd`, `bvsub`, `bvuadd`, `bvumul` and so on. Other operations include `concat` to concatenate two bitvectors, and `extract` to extract a smaller bitvector from another.

The theory of arrays represents arrays in such a way that values can be stored and retrieved from each index. A `store` function takes two arguments and returns a new array with the given element stored at the passed position. Subsequent `select` calls on that position and array do then return the previously stored element.

```
; Bitvector Example
(define-fun is-power-of-two ((x (_ BitVec 32))) Bool
  (= #x00000000 (bvand x (bvsub x #x00000001))))

; Array Example
(declare-fun a1 () (Array Int Int))
(assert (= (select a1 x) x))
(assert (= (store a1 x y) a1))
```

Figure 2.2: SMT-LIB Example of the Bitvector and Array Theory

In Figure 2.2 we first see an example of a definition of a bitvector function that asserts that its 32 bit parameter is a power of two. The shorthand `#x00000000` describes the hex representation of a 32 bit bitvector equal to the number 0. A second example shows the definition of an array, and two instructions asserting the functionality of the `select` and `store` functions.

[3]http://smtlib.cs.uiowa.edu/

3

Related Work

The foundation of this work is based upon several advancement in research performed previously. In this chapter we present an overview and short history of the field of program testing techniques relevant to this thesis. We further describe the feature sets of market leaders in the different areas of fuzzing engines, including the current state of art in symbolic execution engines.

3.1 Fuzzing Engines

The term “fuzzing” was introduced by Barton P. Miller in 1988 and is described as the act of executing a program with (semi-)random inputs in an automated fashion with the goal of causing errors in the program. Pure randomized fuzzing has in the past been the base of a number of studies and has led to discoveries of many problems in program suites [MFS90, MCM06, FM00, KKS98, MKL+95]. Its applicability is however limited, as most inputs would not lead to very deep program paths. This can be easily illustrated with the example of a program expecting certain magic bytes at the beginning of a file. A single four byte magic value would have an incredibly low chance of one in 2^{32} to be correct in full random fuzzing.

More modern fuzzers either depend on grammars to generate better and valid inputs, or are started with a number of already valid seed inputs. Further inputs are then generated by only randomizing parts of a valid input file (mutation-based fuzzing), or to use basic program instrumentation techniques to measure how much of a program is covered by an input. Quality inputs can then be used to generate new, better inputs (generational-based fuzzing). Other methods include the use of pre-defined protocol grammars [Ait02, GKS05] to generate more efficient inputs

at the cost of additional work required on the side of the developer to define the grammar.

American Fuzzy Lop[1] (AFL) is an advanced open-source automated fuzzing suite that applies a number of these concepts. Its main features are its speed, reliability and simplicity. By applying a number of preprocessing and mutation strategies, it is able to achieve high coverage and has discovered a number of bugs in image processors and parsers. By trading in expensive analysis techniques such as symbolic execution for speed, it can generate high quality inputs. The common problem with magic numbers however remains, and AFL is generally unable to get past those without high quality seed inputs to start off with.

Other work has been done using information flow techniques to track which bytes of the input can affect the control flow inside the program [BBGM12, LBB+07]. These approaches can fall into the category of hybrid systems between pure black-box fuzzing, where no knowledge about the program internals exists, and more advanced techniques involving symbolic execution.

3.2 Symbolic Execution

One of the technologies to generate smarter test inputs for software systems is to utilize symbolic execution to derive constraint formulas on the input. While the idea behind it is an old one [Kin76, Cla76, RHC76, How77, BEL75], it has only recently again caught interest in the area of program testing. Due to this, there are no big, open source projects comparable in ease of use to American Fuzzy Lop.

A large open source contender in the field is KLEE [CDE08]. Here the authors extend on the previous work on the similar EXE [CGP+06] tool. KLEE works on LLVM bytecode, by adding custom instrumentation to it that is used to perform symbolic execution on designated inputs. As such, it generally requires the source code for applications, so that they can be compiled to LLVM bytecode with, for example, Clang [Lat02] in the case of C/C++. KLEE's two primary goals are to (1) cover every line of code in the executable program and (2) detect operations and bugs that could be exploited. As a pure symbolic execution toolkit, KLEE executes every branch simultaneously and thus faces the problem of exponential path explosion. The authors have however developed more efficient caching structures and search heuristics in order to prioritize interesting execution paths, and have successfully used their tool to detect tens of bugs in the well-tested GNU Coreutils tool package, amongst others [CARB12].

An interesting development on top of KLEE is the S^{2}E [CKC11] project. By developing a module for QEMU that effectively translates assembly languages

[1] http://lcamtuf.coredump.cx/afl/

to LLVM [CC10], the authors are able to provide symbolic execution even for proprietary software systems, and, more importantly, to correctly symbolically execute operating system and driver APIs instead of just emulating their behavior. On top of that the authors's main contributions are (1) selective symbolic execution to limit expensive analysis to interesting parts of the system, and (2) execution consistency models to reduce the computational overhead when it is not required. The efficiency of the tool depends largely on custom-written analysis plugins, that reduce the number of execution paths down to an interesting or relevant number, thus allowing the execution to reach deeper execution paths before running out of resources.

The described projects all perform full symbolic execution, meaning that at each branch they have to fork off a separate execution, effectively doubling (ignoring shared resources) the resource load.

S^2E attempts to limit the impact of this exponential path growth, by providing a plugin architecture that can prioritize exploration of certain paths and limit the effect of the inherent path explosion of symbolic execution. It does however mean, that certain executions will potentially never be explored. Since these path selection plugins work with heuristics, a wrong guess can lead to missing important bugs.

Following we describe some projects that attempt to circumvent this limitation.

3.2.1 Concolic Execution

The "Scalable, Automated, Guided Execution" [GLM+08] (SAGE) toolkit, is Microsoft's research into applying symbolic execution to automatically generate test cases for fuzz-testing of applications. The authors introduce a method known as "whitebox fuzz testing" which effectively uses **concolic execution** to symbolically execute a single exeecution trace that resulted from a single input. With this, SAGE is able to scale to much larger applications, without missing out on any potentially vulnerable branches by dropping them to save resources.

After capturing the branch constraints of the execution, SAGE then uses them to generate inputs for new executions. It uses a method called "generational search", to intelligently pick inputs that will have the highest chance of leading to uncovered code. Aside from its scalability, SAGE has the advantage of running on X86 assembly, and thus being able to detect errors that manifest themselves during compilation or other build processes.

Microsoft has used SAGE to discover a number of possible high-profile bugs, notably about one third of all the file-system related bugs discovered through fuzzing during the development of Windows 7 [God09]. Since SAGE is an internal toolkit at Microsoft, it has since not been released to the public, neither in source nor binary form.

With CREST [BS08], the creators built a symbolic testing program, that uses the C Intermediate Language (CIL) [NMRW02] to perform source-to-source transformation and insert instrumentation functions into the C code. CREST then performs concolic execution of the instrumented program. Its main contribution is the introduction of a number of new search strategies for exploring the space of possible execution paths more efficiently.

In addition to the two extremes of pure symbolic execution and pure concolic execution, other hybrid variations exist.

3.2.2 Hybrid Execution

With FuzzBALL [BMS11, MPC+13, CBP+13] the authors combine static and dynamic analysis techniques to intelligently pick the locations of potential vulnerabilities inside the program. They then use symbolic execution to direct the path exploration towards the potential target vulnerability. FuzzBALL works on X86 assembly and the authors have recently made its source code public on GitHub[2].

Mayhem [CARB12] is one of the newest generation of symbolic fuzzing frameworks. Similarly to SAGE, it performs its analysis on X86 code. Its main features are comprised of scaling online symbolic execution to large applications, while keeping its speed compared to offline (concolic) execution, and a more accurate representation of symbolic memory accesses. Mayhem can create test cases for an application, discover possible bugs and automatically generate shell-spawning exploits. Its scalability is achieved with "hybrid symbolic execution": Online symbolic execution is performed of the program until the memory pressure of the system becomes too high, at which point Mayhem creates so-called checkpoints for some execution states that contain all path constraints and additional information required in order to restore the execution state for that point. Once execution finishes, these checkpoints are restored by concolically executing the program with the saved path constraints to reach the previous state, at which point online execution continues.

The authors of Mayhem ran their tool on almost every binary of Debian Wheezy, generating about 1200 bug reports[3]. Most of the generated reports concern minor issues and do not directly cause security issues, however. Like SAGE, Mayhem remains an internal toolkit and the authors are working on providing a commercial testing service using it[4].

[2] https://github.com/bitblaze-fuzzball/fuzzball
[3] http://lwn.net/Articles/557055/
[4] http://forallsecure.com/mayhem.html

3.3 Summary

We can see that the area of discovering program errors is rich with a wide range of techniques and approaches, however a common trend of the existing research and tools remains to only focus on the common x86 processor architecture. This effectively rules out the application of any of these results in the area of embedded systems development without significant further work. The extent of this work depends on each project, but at a minimum consists of modeling the effects and side-effects of additional assembly languages.

Additionally, even for toolkits that do claim multi-architecture support, they remain proprietary or depend on the program's source code in order to create proper inputs. It has been shown however, that many potential errors only manifest in the build-chain, or in (potentially) closed-source dependencies [CKC11, CARB12, BMS11].

As such, we focus on bridging this gap with our work. We decided to utilize the advantages of PANDA's record and replay system in circumventing application timeouts during expensive analysis, in order to perform concolic execution of a record to generate domain-specific input files to test the program with. This essentially gives us X86 and ARM support for free, with further architectures (X64, MIPS, ..) available with limited extra work. Additionally, the implementation of symbolic execution operates on LLVM, and thus remains detached from the CPU architecture.

When looking at some of the top related work in our field, such as SAGE and Mayhem, it might become apparent that fuzzing using symbolic execution is already an advanced and nearly magical method. In contrast, the results of our work might seem insignificant. However, we believe there are two primary issues with that stance, that are largely in-sync with the results of a discussion started by the developer of AFL[5].

First and foremost, the programs developed in the most popular contenders are largely non-public. This makes it impossible for anyone to reproduce the results and thus confirm them. The developers of SAGE claim it is used internally at Microsoft, on internal tools. No bugs discovered by it are made public, and anyone reading the related publications has to take the authors's word for the success. Mayhem has become popular due to the impact it caused by releasing thousands of bugs discovered in the GNU coreutils suite, however after investigation, nearly all of these are in insignificant, non-critical areas, that could hardly be used to compromise a system. The authors do not talk about the important limitations their tools have, and since they remain non-public, it is not possible to determine whether the tools would be applicable to real-world systems.

[5]http://lcamtuf.blogspot.se/2015/02/symbolic-execution-in-vuln-research.html

Secondly, a number of authors in popular research in this field compare their results to that of completely random input generation. While this is a good way to establish a baseline, it does set the bar to reach incredibly low, as being better than random mutations is almost guaranteed for any technique which is moderately novel.

Consequently, we decided to compare our work to that of American Fuzzy Lop. While AFL is a state-of-the-art fuzzing engine that has received significant work and results of extensive research[6], we chose it primarily because of its ease-of-use and easy availability. We do not expect to beat AFL in all cases, but we do believe it is a good measure of how close we can get. During our evaluation in Chapter 5, we see that our solution even outperforms AFL in some cases.

[6] http://lcamtuf.blogspot.se/2014/08/binary-fuzzing-strategies-what-works.html

4

Design and Implementation

This chapter describes the design of the system developed to maximize code coverage, and the difficulties solved during implementation. We describe the general structure of the approach and then go into detail of the individual steps involved, the problems encountered and their solutions. We use the more scalable variation of symbolic execution, concolic execution, in order to intelligently generate new program inputs that target unreached parts of the program. Also, for reasons discussed in the introduction, the primary goals, aside from coverage maximization, remain the following:

1. No access to program source code of the tested application.
2. Support for multiple architectures, such as ARM and X86.
3. Zero knowledge about the internals of the program.
4. Unsupervised execution of the system while it fuzzes the target program.

Whether the resulting system fulfills the goals, and how efficient it is compared to blackbox fuzzing is then evaluated in the next chapter.

4.1 System Structure

The coverage frontier [LBB+07] describes the branch conditionals for which only a single branch of the two available ones (true and false) have been taken. The general goal when trying to maximize coverage, is to target branches on the coverage frontier. By using specially crafted input files the execution is directed towards

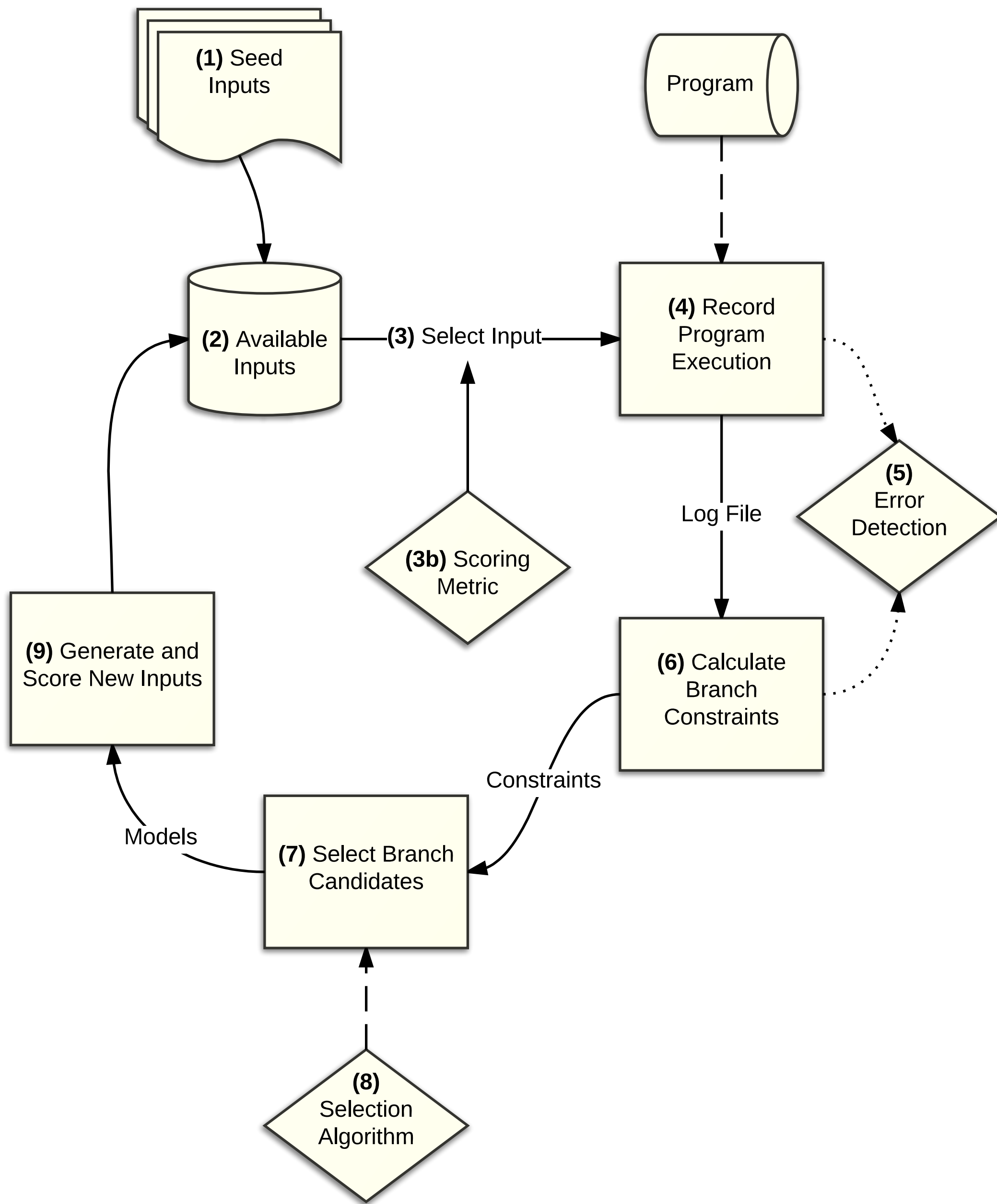


Figure 4.1: Flowchart of the project's structure

these branches, leading to the discovery of new basic blocks and new (uncovered) branches thus extending the coverage frontier.

Our approach is separated into a system of several individually performed steps. Figure 4.1 shows a general overview of the architecture. We will now outline the responsibilities of each step:

(1) Initially, the algorithm needs to be initialized with a number of seed inputs. These inputs are used to perform initial runs of the target program. The quality of the inputs (i.e., how domain-specific they are) can greatly increase the effectiveness of the algorithm, however, completely random inputs are perfectly valid.

(2), (3), (3b) At this point the fuzzing loop of our work begins, and a collection of inputs available for testing is constantly kept. Each input is associated with a score that is meant to give a hint of how high the potential for newly discovered branches is. The input with the best score is then selected for further analysis.

(4) The input and the target program are prepared to be executed and a trace of the execution is recorded with PANDA. During both this stage and the next, potential execution errors (bugs) in the target program can be detected by a different system (5).

(5) At this point it becomes possible to detect crashes, memory errors and potential security vulnerabilities in the code. We do not handle this part in this work.

(6) The heart of our approach: Using PANDA's replay mode, the previously recorded trace is executed again and bytes read from the input file replaced with symbolic variables. During replay, instructions operating on the input file are then executed symbolically (concolic execution). For each branch instruction depending on the input file, the symbolic expression is then output as a constraint placed on the input by the branch condition.

(7), (8) After all constraints have been captured, some of them are selected to be negated using a selection algorithm in (8). The new modified path constraints are then attempted to be solved by a SMT solver, and for those that are satisfiable, models are generated.

(9) The models from the previous stage are used in order to generate new potential inputs based on the original input file and are scored accordingly using a scoring metric algorithm (10).

In the following sections, we will go into more detail of the stages and the challenges involved in them.

4.2 Input Files

The input files, or test cases, of the program to be tested are the target of our system. They get mutated into new versions using the constraint information captured during the analysis phase. Initially, the system needs to be seeded with a number of **seed inputs** that are used to perform initial runs of the program.

While randomly generated seed inputs are acceptable, it makes sense to provide a number of different length seed inputs. This is because our system only varies the content of the file, but not the length of it. An input of ten bytes will always remain ten bytes long, even after being mutated by our algorithm multiple times. It could then be possible to never reach certain code locations that depend on the physical length of the input file.

4.2.1 Scoring

Each analysis phase can, depending on the number of captured constraints, produce a large number of new input files. A large number of those will most likely not advance the coverage frontier any further. Since the analysis stage takes significantly more time than that of a blackbox fuzzer such as American Fuzzy Lop, we must take extra care in picking a good input.

Test cases are sorted with a scoring metric using the following values, prioritized in order:

1. Targets the uncovered edge of a known branch.
2. Produces a new path of execution.
3. Number of inputs executed targeting this branch (less is higher priority).
4. Length of the path (shorter is higher priority).

These criteria have been determined through common procedures [GLM+08] such as prioritizing uncovered branches, and through testing of different selectors. Ultimately the current set of criteria has proven to be the best performing, however it is possible to vary them further.

First, selecting an input that covers an uncovered branch of a conditional instruction is guaranteed to cover new code, as such it has the highest priority. Second, there is no use in executing an input that would produce no new execution path, as it does not reveal any new information, therefore those inputs that do

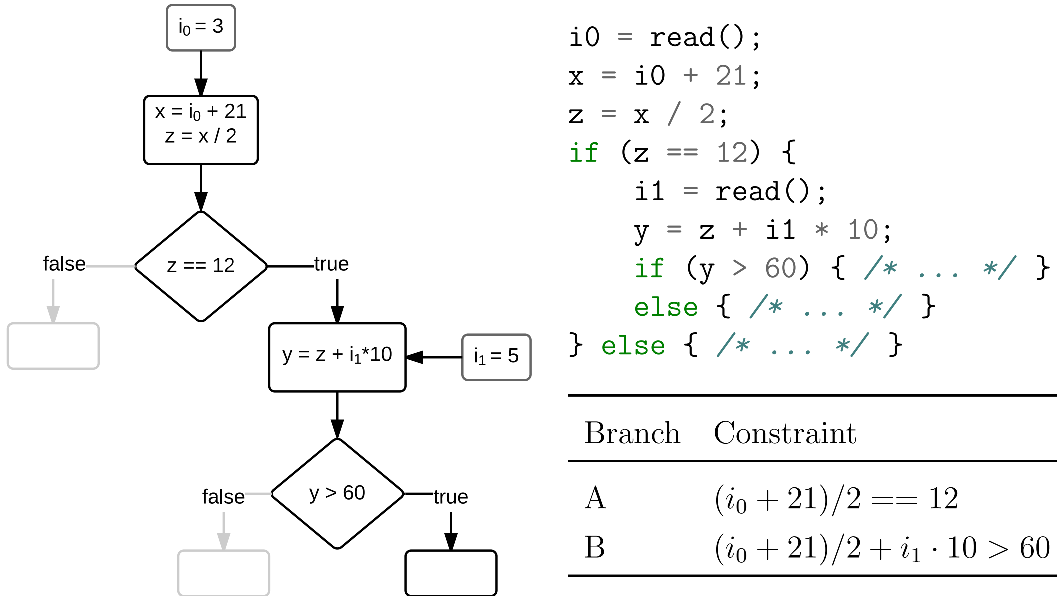
lead to new execution paths are prioritized. The next two selectors are primarily the result of experimentation.

4.3 Recording

After an input file has been picked, it is packaged together with the application in an ISO file. A QEMU instance is started and resumed from a snapshot in order to avoid a lengthy boot process. The ISO file is mounted in the system, then PANDA's recording functionality is activated and finally the program executed. After execution finishes, the recording is stopped and the system shut down. The resulting execution trace is then used for the following concolic execution.

4.4 Branch Constraints

The largest and most important part of this work is the construction of branch constraints on the input file that govern the execution path and thus parts of the code it will reach during execution.




```
i0 = read();
x = i0 + 21;
z = x / 2;
if (z == 12) {
    i1 = read();
    y = z + i1 * 10;
    if (y > 60) { /* ... */ }
    else { /* ... */ }
} else { /* ... */ }
```

Branch	Constraint
A	$(i_0 + 21)/2 == 12$
B	$(i_0 + 21)/2 + i_1 \cdot 10 > 60$

Figure 4.2: An example of concolic execution with branch constraints in our implementation. The pseudo code example on the top right with its control flow graph on the left, and the extracted branch constraints in the bottom right.

In Figure 4.2 we see an example of symbolically executing a single program trace. We designate the variables i_0 and i_1 as inputs and treat them as symbolic

variables. Since we symbolically execute an existing program trace (i.e. concolic execution), we do have concrete values available (3 and 5 respectively). Therefore at branch A the value of z is pre-determined and the execution takes the true branch. At this point we output the constraint that this branch enforces on the input. Concretely, this means that any execution of the program, that should take the true branch of this conditional, requires the branch A constraint to hold. When the execution reaches condition B and takes the true branch, we output a second constraint on the input. Now, for any execution that intends to reach this exact point, the branch constraints A and B have to hold for the input.

Using these branch constraints, it will then be possible to generate an input file that leads a new execution down a different branch. We will now detail the challenges involved in calculating these constraints from PANDA's LLVM code representation.

4.4.1 LLVM Instrumentation

To be able to extract runtime values from the target program and identify operations for symbolic execution, it is necessary to augment each line of LLVM code with special callback functions. This technique is generally called instrumentation or code tracing. It is implemented by a custom LLVM pass, that is executed on every basic block translated by QEMU/PANDA. The pass inserts the necessary tracing calls for both the taint flow and our symbolic execution system. For a binary operation (`add`) this would, for example, look like the following excerpt:

```
%17 = add i64 %16, 1
call void @taint_mix(...)
call void @symbexec_binop(...)
```

Here `taint_mix` propagates the basic taint information for the addition and `symbexec_binop` records the more detailed result of symbolically executing the two operators. A symbolic execution callback is not always necessary, as the taint system is capable of handling, for example, a pure copy between two LLVM registers.

It is worth noting that LLVM remains an **abstraction** of the assembly that's actually being executed during runtime. The LLVM registers do not directly represent locations in memory. The shadow memory implementation of the PANDA taint system, however, does assign each LLVM register, LLVM return value and others a unique location in the corresponding shadow memory. This does not affect the correctness of the LLVM representation of its originating TCG IL from QEMU, it merely adds an indirection during taint tracking through the introduction of LLVM registers. For example, while an `add` instruction originating in assembly is a single instruction, the corresponding representation in LLVM IR

Name	Purpose
RAM	System memory
I/O	I/O memory
LLVM	LLVM variables of the current function
RET	LLVM return value
GRV	CPU registers
GSV	Special CPU state flags (EFLAGS, FPU, etc.)

Table 4.1: Different types of shadow memory. We list the number of shadow memory instances for the various emulator locations that we track taint flow through.

consists of two load operations storing the operators in LLVM registers, an LLVM add instruction storing the result in a new LLVM register and lastly a store of the register value into the destination register of the original `add` instruction.

We based our solution on an existing taint tracking plugin in PANDA [DGLHL13, WLK13]. This plugin is a rewrite of an earlier, slower one from the developers, and has a number of different shadow memories for different physical and logical memories. Table 4.1 shows the different types of shadow memory used to track taint across system execution.

During development, we discovered a number of errors in the implementation that became particularly visible during symbolic execution. Undertainting in particular can cause a number of divergences from the program source. The errors we corrected are, among others:

- Lack of taint tracking through I/O memory.
- Incorrect tainting of LLVM function call arguments.
- Loss of taint on LLVM allocated stack variables.
- Use of unallocated variables during instrumentation leading to insertion of erroneous calls.

As of the writing of this thesis these corrections have not been merged back into the original project yet, however the developers have been informed of the errors and work is being done on integrating them into the source repository.

4.4.2 Memory I/O

Values read from and written to memory need to be instrumented and specially mirrored to shadow memory. We need to capture potential symbolic values read

from (or written to) shadow memory, while still mixing in concrete values from the underlying storage if no symbolic values are present.

To achieve this, whenever an instruction reads a value from memory as an operator, and that memory is marked as tainted, we construct a new bit-vector expression with the size of the read. The expression is constructed by iteratively concatenating one of two bytes at the memory location of the read: If the memory location is not tainted, the byte from the concrete memory is picked and turned into a bit-vector. However, in case the read location contains an expression, that expression is concatenated instead. Values of multiple bytes are considered stored in common least significant byte (LSB) first order therefore, the value has to be concatenated in reverse.

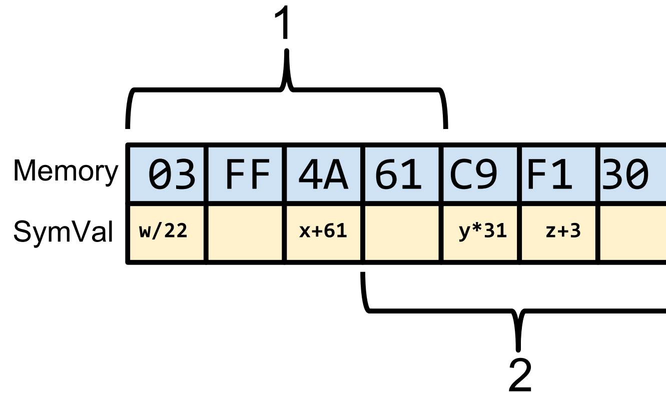


Figure 4.3: Example of two memory reads, showing symbolic and concrete memory

Two examples of read operations are shown in Figure 4.3. Operation one performs a read on the first four bytes, containing a total of two symbolic and two concrete values. As the value is constructed from left to right, LSB first, it results in the following SMT-LIB expression: `(concat #x61 (concat (bvadd x #61) (concat (concat #xFF (bvsdiv w #22))))`. The concrete bytes `0x03` and `0x4A` are thus ignored, since symbolic expressions are available for these. Operation two similarly results in an expression of `(concat #x30 (concat (bvadd z 3) (concat (bvmul y 31) #x61)))`.

Writes to the shadow memory happen in a very similar way, but here, instead of concatenating bytes together, the full multi-byte expression is available and has to be split into its individual bytes before being written. This is done using SMT-LIB's `extract` instruction and the example in Figure 4.4 shows the result of that operation. It is worth noting that all writes at this point refer to writes to the shadow memory, i.e., the data structure used to track the flow of information.

While Figure 4.4 shows the end-result of the write, the expressions temporarily become much more complex, due to the repeated concatenation and extract

`(concat (bvadd x #33) (concat #xA3 #x81))`

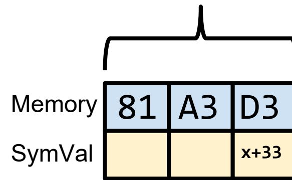


Figure 4.4: Example of a symbolic write to memory

Byte	Expression	Simplified Expression
0	`(extract 7 0 (concat (bvadd x #33) (concat #xA3 #x81)))`	`#x81`
1	`(extract 15 8 (concat (bvadd x #33) (concat #xA3 #x81)))`	`#xA3`
2	`(extract 23 16 (concat (bvadd x #33) (concat #xA3 #x81)))`	`(bvadd x #33)`

Table 4.2: Result of extracting bytes from the expression in Figure 4.4

processes. Table 4.2 signifies the importance of Z3's `simplify()` function for expressions: The table shows both expression versions, the one before the simplification, after multiple concatenations and extractions, and the much shorter expression after simplification. Intuitively, this becomes even more important in a real application with hundreds or thousands of I/O operations.

4.4.3 Operator Translation

As the most important part of symbolic execution, it is necessary to correctly mirror each LLVM instruction in the form of a symbolic SMT-LIB expression. To properly perform this, the following information needs to be passed on during instrumentation:

1. type of backing shadow memory
2. source operands and destination address and size in shadow memory
3. concrete values for all source operands
4. type of instruction

Most of LLVM's operations translate directly to SMT-LIB format. Both have support for unsigned and signed operations of addition, subtraction, division, multiplication, remainder and all binary operations on bit-vector integers. Floating point operations remain unhandled for the moment. Similarly, for comparison instructions, SMT-LIB instructions exist for unsigned and signed variants of all inequality operations.

LLVM however has a few so-called intrinsic functions that perform common operations that can not be modeled with existing instructions, or would require too many instructions. These functions are normally translated to corresponding assembly instructions by LLVM's compiler backends, however in our case we have to model their operations on top of symbolic memory. Only two operations were required to be modeled in PANDA: `uadd_with_overflow` performs an unsigned addition and produces an extra bit designating if an overflow occurred in its result type `{ i1, i32 }`. `bswap` performs a byte swap of its single operator, effectively reversing the bytes of it.

4.4.4 Symbolic Pointers

A special case of memory I/O is when the address read from or written to is tainted, i.e. is a symbolic value. This involves that the exact result of the operation depends on the input. This is a common case in array lookups, or lookup tables in general and needs to be handled. We currently handle the most common event, symbolic pointer read operations, by using the theory of arrays for SMT. In the event of such a memory read, we first determine the bounds of the pointer expression by using the SMT solver to iteratively refine the minimum/maximum value of the expression. Once the bound is determined, an array expression is created and all concrete memory values present at the possible pointer locations are stored inside it. To finalize the resulting expression, an array theory read operation at the location of the pointer expression is then performed on the created array expression. The resulting expression can then be as an accurate modeling of the symbolic pointer read and used in further constraints.

4.5 Test Case Generation

In the final phase, once the constraints of the current test case have been captured, new inputs are generated using the solutions to the constraints produced by Microsoft's Z3 SMT solver. The conjunction of all captured constraints during symbolic execution are called a path constraint. In order to generate new inputs that will lead the executed branches into different directions, those individual constraints are iteratively negated, as can be seen in `AnalyzeLog` of Figure 4.5. The

process is very similar to that of SAGE's [GLM+08] generational search. For each new test case the depth at which it has been generated at is kept as a lower bound for further children to be generated from this test case. This is to prevent redundant creation of already existing children from further up in the path constraint.

```
AnalyzeLog(input, log) {
  PC = symbolicallyExecute(log)
  for (i = input.bound; i < length(PC); i ++) {
    newPC = PC[0 .. i-1] + not(PC[i])
    if (newPC is SAT) {
      newInput = input + model(newPC)
      newInput.bound = i
      AddInput(newInput)
    }
  }
}
```

Figure 4.5: Pseudo code showing how new test cases are generated from a program recording and its corresponding previous input.

In the example of Figure 4.2, this process would result in two new inputs. Here, the original path constraint consisting of the two branch constraints A and B is expanded to `not(A)` and `A, not(B)`.

5

Evaluation

In this chapter we evaluate the developed system with respect to the set goals, its efficiency and comparison to blackbox fuzzing. Initially we describe the methods and metrics used in this chapter, then we elaborate on the individual tests performed. Finally we present the test results.

5.1 Methodology

To determine the effectiveness of our system, we compare it to a sophisticated conventional blackbox fuzzing engine, American Fuzzy Lop (AFL). Since it is infeasible to measure coverage for each executed input for AFL, because it executes thousands of test cases per second, AFL outputs test cases every time a new program path is executed. Using a third-party tool named `afl-cov`[1], which uses `gcov` to determine line coverage, we also determine increasing line coverage with each test case.

Our goal is to test how our implementation handles complex programs. Complexity can appear in different forms. Here, we handle it in two cases: First, we consider complicated branch conditions, such as equations in integer arithmetic or comparison against certain magic values that are hard to guess. Secondly, complexity can appear in the form of the size of the code and the number of control flow construct, loops in particular. This case in particular tests the ability to scale to larger applications and handle the problem of non-terminating loops.

Both our solution and AFL are allowed to run for up to two hours after no change in coverage has been observed or until all reachable locations are covered. For AFL this means the process is canceled after the creation of the last test case

[1]https://github.com/mrash/afl-cov

was more than two hours ago, for our solution it means no change in line coverage has occurred for more than two hours. Our tests and results later show that this relatively arbitrary time does not favor our solution over AFL, because if AFL manages to cover a code location for our small program samples, it does so within seconds to minutes.

After the time has expired, the process is aborted and the coverage results are extracted. We then compare the number of lines covered over the number of inputs and over time. For some of our tests, we perform a more detailed analysis to determine if certain conditionals have been covered.

The measurement of code coverage becomes a challenge on its own, due to the Assembly -> TCG -> LLVM recompilation performed by PANDA. PANDA, together with QEMU, performs dynamic recompilation of the source binary into LLVM (and later the host system's architecture), as such only code locations visited by the program are actually visible to PANDA. This poses a difficulty in determining the maximum number of basic blocks, which is a necessity in order to derive a total percentage of covered code. This is further complicated, because of QEMU's helper functions (i.e. for divisions) introduce additional branches and basic blocks that could be influenced by input data.

Due to this reason we have decided to use `gcov`[2] to measure the coverage during each run. `gcov` is linked into the binary during compilation, and results in a trace file being created each time the binary is executed. Using this data, we can determine the number of basic blocks of the original program covered during a run. There are a number of limitations to this: First, `gcov` can not capture coverage if the application crashes (because of, for example, a division by null), however, proper crash detection is an additional challenge that we do not handle in this work. Secondly, only coverage of the C sources of our test applications is captured. This does not include the mentioned QEMU helper functions, nor any use of library (such as libc) functions where file taint could reach further branches. However, since we generally assume that source code is not available anyway, and our test cases are strictly controlled, this remains an acceptable inaccuracy for our evaluation.

For each test we place the precompiled binary into the folder that will be provided to the QEMU virtual machine and modify the `run.sh` script to start the target executable. A single seed input consisting of several repetitions of a single character is used. After fuzzing is finished, we use the target executable, compiled with `gcov` coverage support, to generate coverage statistics for each executed test case.

[2] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

5.2 Tests

We separate the performed tests into two categories: The first (from now, prefixed with `t`) contains a number of tests that attempt to validate the correctness of the symbolic execution implementation and its performance against complex branch conditions in isolation. It does so by performing a number of arithmetic calculations and comparing their results, and presenting loops that require several iterations in order to generate the expected result that lead to further uncovered code. In order to compare the results better, we also note if AFL managed to generate test cases that cover the target of each test.

The second is a number of samples from real-world applications. This involves larger programs with more complicated logic and loops running for several iterations. This is to test if our solution can handle the combination of complex branch conditions combined with multiple iterations of loops and other execution flows indirectly influenced by the input. We use the test suite from previous work on analyzing static program checkers [ZLL04]. The suite has been used in a number of other works on program testing to varying degrees [LBB+07, Zit03, ZLL05]. Unfortunately not all of the samples could be run with our implementation. This was either because the samples were not easily convertible to our dynamic analysis technique (due to non-deterministic side-effects such as network activity), or because they hit other current limitations in our implementation, such as scalability issues with the Z3 SMT solver among others.

5.2.1 Complex Branch Conditions

We begin by describing the results of the tests in the first category. Each test has one or more **target branch conditions** which represent a category of complex operations. The goal for each test is for both our implementation and AFL to cover them in as few test cases as possible.

t1 Simple Comparison: Three nested `if` statements comparing input bytes to constant values.

t2 Simple Arithmetic: A number of arithmetic operations to verify common binary operations (division in particular). Operations are performed on two bytes which are read from the input file.

t3 Loop: Usage of `atoi` converts a string read from the file to an integer, and compares it to a constant value. The target branch condition is `atoi(input) == 31337`.

	Covered by		
	AFL	Ours	Inputs needed by our solution
t1	Yes	Yes	4
t2	Yes	Yes	12
t3	No	Yes	22
t4	No	Yes	6
t5	No	Yes	5

Table 5.1: Results of the performed tests. Our solution, as opposed to AFL, managed to cover all branches. We also list the number of inputs our solution had to generate to achieve full coverage.

t4 Complex Arithmetic: A larger equation based on two integers read from the input file is compared to a constant value. Target condition (with a, b as four byte integer inputs) is `(a*a + 512*b - 912) / (b - 5 * a) == 31337`.

t5 `strcmp`: A constant string compared to the input using libc's `strcmp`.

We ran those tests in AFL and our system. For AFL, we noted if it managed to cover all the branches within the time limit of two hours, for our system, we noted the number of inputs it took to cover the target branch conditions.

We can see from the results in Table 5.1 that our solution performs exceptionally well with the more complicated branch conditions. Even the slightly more difficult `t3` depending on `atoi` is solved after 22 inputs.

Both `t1` and `t2` have two one byte integers as input, which provide a state space of $2^8 + 2^8 = 512$ for naive bruteforcing. Since AFL is executing ten thousands of input variations each second, it quickly solves the branch constraints present in the tests. Its limits become clearly visible with `t3` to `t5`, where more complex conditions are present and our solution easily solves them within the minimal amount of executions.

However, AFL still outperforms our solution on the simpler tests operating on two one byte integers. Of note is, that if AFL managed to cover all target branches, then it did so after a few seconds.

5.2.2 Complex Control Flow Logic

The second category consists of the following three tests. The entire static code checker test suite consists of fourteen tests from the previously mentioned test suite

for static code checkers [ZLL04]. Out of the remaining eleven tests not present in the following list, one third was not applicable to input file fuzzing as outlined in Section 5.2, or consisted of very similar variations of other tests. The remaining tests had problems with our unoptimized handling of pointer arithmetic and got stuck during SMT solving of the constraints. For the three samples we kept the names from the original test suite: `crackaddr` (also called `s1`), `mime1` (`s3`) and `prescan` (`s5`). All three samples are between 500 and 600 lines of code (LOC), including comments. In the following results, we only measure the *reachable* LOC, also called *effective* lines of code, or ELOC.

We plot the results of both our solution and AFL both as a function of execution time, and as a function of the number of generated test cases. Test cases in AFL require some clarification: While AFL executes hundreds of thousands of different inputs per second, it only stores inputs that cause the program to exhibit new behavior, i.e. a new execution path. AFL also calls those "test cases". As such, each test case from AFL can correspond to a large number of actual program executions and corresponding generated inputs.

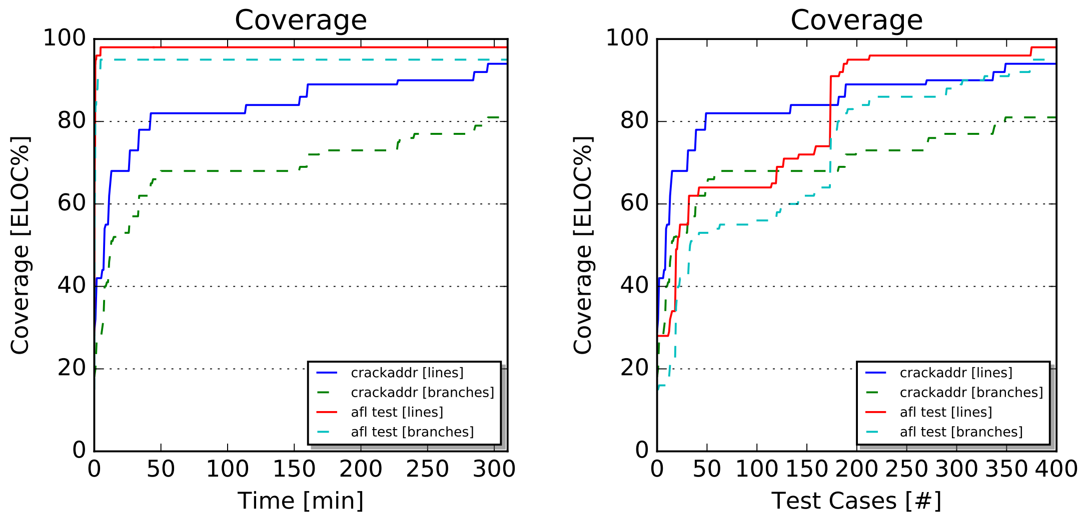


Figure 5.1: `crackaddr` test results. In the tested time, our solution does not reach the same coverage as AFL. Over the number of test cases, our coverage stays superior than AFL for a while.

The results are plotted in Figure 5.1, 5.2 and 5.3. The graphs are truncated shortly after the maximum in coverage was reached until the previously described two hour timeout. In Figure 5.1 this resulted in a total execution time of nearly seven hours, because our solution always managed to cover more lines before the chosen timeout has expired.

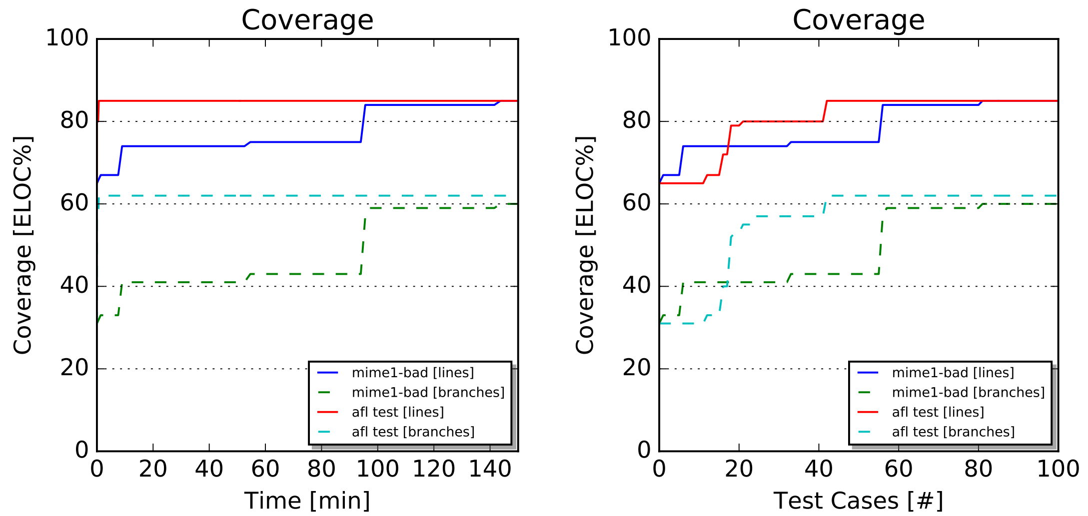


Figure 5.2: `mime1` test results. Our solution reaches full coverage after roughly two and a half hours. Results over number of test cases are similar.

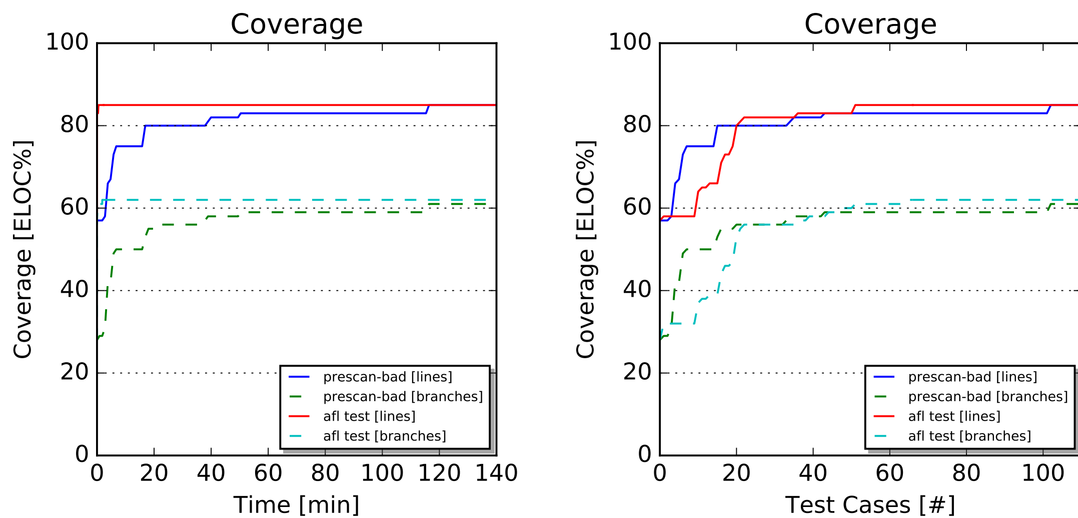


Figure 5.3: `prescan` test results. Full coverage is achieved after two hours. Again, results over number of test cases are similar.

	Ours (s)			AFL (s)		
	Average	Median	StdDev	Average	Median	StdDev
crackaddr	51.0	50.0	3.4	5.7	0.3	29.0
mime1	123.0	125.0	24.7	36.0	0.4	184.0
prescan	64.0	59.0	12.0	2.0	0.1	5.0

Table 5.2: Execution time per generated test case

The maximum coverage primarily differs from 100% due to the presence of unreachable lines of code that `gcov` counts towards the total. This means that any branches leading to those lines are unaffected by the input and could not be reached by either AFL or our solution.

We can see from the results that our solution performs much worse than AFL in the time dimension. This is not unexpected, as AFL is a matured software developed on a large number of well-known and well-tested techniques. While AFL reached its maximal coverage within a few minutes for all three cases, our solution took several hours to reach the same (or similar, in the case of `crackaddr`) percentage.

The results look different if plotted in comparison to the number of inputs. Here our solution shows much clearer that it produces test cases of at least as good quality as AFL. It is however also noticeable where our solution has to effectively blindly try to explore conditionals in order to discover new, uncovered, conditionals. This becomes visible in the form of plateaus, where the coverage does not increase. Since we do not perform analysis of the program's control flow graph, we can not detect which conditionals would advance an internal loop, or reveal a nested, uncovered, conditional. This is primarily a result of our simplistic scoring algorithm, which we did not focus on. The algorithm can be further improved in a number of ways, as expanded upon in Chapter 6.

Furthermore, we look at the execution time results in Table 5.2. We show the average and median time taken between produced test cases for our solution and AFL, along with the standard deviation. While it was clear from the beginning that AFL would perform significantly better, we still provide the time it took to find new test cases for completeness's sake. We see that the execution time for our solution varies with the executed sample, but remains relatively constant. It is not viable to interpret these numbers as execution time per program run, since there are thousands of actual executions between generated test cases in AFL, while ours produces a new test case with each execution.

6

Future Work

As became clear from the results, our work is merely a stepping stone towards a full binary symbolic fuzzing engine. In this section we describe potential future work to improve the results.

6.1 Error Detection

An important part of fuzzing aside from high quality input is the actual detection of errors in the program. Since those errors do not always manifest in the form of very obvious program crashes, or return codes, detecting them is a challenge on its own that we did not address in this work. Future work could investigate analysis of the program's runtime during replay or recording as to undesired side-effects.

One category of well-studied errors is that of use-after-free or buffer overflow bugs. To detect those, there exists software libraries such as Address Sanitizer[1] which could be possible to integrate into the tested executable. The project implements forms of taint-tracking and shadow memory techniques to detect access to unallocated or freed memory locations. Since this brings additional overhead into the analysis, it would be best to perform this as a parallel step on the already available input file. Address Sanitizer currently requires the LLVM source in order to add instrumentation to the executable, however it could be possible to integrate this directly into PANDA's LLVM representation on a block level. A different choice might be Valgrind[2], since it does not depend on the availability of the source and it is a well-tested, matured toolkit.

[1] http://clang.llvm.org/docs/AddressSanitizer.html
[2] http://valgrind.org/

The Mayhem [CARB12] framework goes one step further, by providing a proof of concept script for each detected exploit that causes the fuzzed program to spawn a shell. While this is an extreme example, the representation of the program's constraints during symbolic execution performed by our work can be further analyzed in order to detect exploits that could lead to control flow hijacking or leakage of secret information. The basis for this analysis is given by our modeling of pointer arithmetic if during a pointer read, the bounds intersect with secret information (which could be tracked through the taint tracking system). Similarly, if during a write to a symbolic address, the bounds intersect with sensitive locations on the stack, such as the return value, the input could be used to hijack the control flow.

6.2 Performance Improvements

The slowdown involved during symbolic execution is not ideal. With runtimes of about one minute for a very small program, the improved test accuracy comes at a big cost, while fuzzers like AFL can test thousands of inputs per second. There have been arguments[3] that this slowdown destroys the usefulness of more expensive analysis techniques, however we believe that this is not true, mostly due to the fact that even relatively advanced fuzzers fail simple magic value checks (we expand on this later).

There are numerous ways to improve the performance of our approach, since we paid little attention to this in our proof of concept. Aside from improvements of the code's performance, a powerful technique is the parallelization of the steps involved in our analysis. Due to the atomicity of most steps, it lends itself perfectly to the problem. For example, a considerable amount of time is spent launching QEMU and setting up the recording environment for each attempt. A better approach would be to keep an instance of QEMU running in the background and delegating the creation of new recordings for inputs to it. Any number of machines or processes can then work on analyzing the recorded log files and generating new inputs.

Currently, our handling of pointer arithmetic is sub-optimal. We replicate large amounts of expressions in order to model the array logic necessary to solve pointer dereferences. A better solution would be to investigate solve strategies and tactics implemented by most SMT solvers (including Z3) that are optimized for a certain amount of duplication in the assertions.

Finally, instead of chosing randomized fuzzing or symbolic execution, it is probable that by combining our technique to continuously generate new and better quality input files with AFL's significant speed, a much faster coverage could be

[3]http://lcamtuf.coredump.cx/afl/README.txt

achieved. Here, if AFL gets stuck in complicated conditionals in the code, our approach will quickly move through it, allowing AFL's techniques to continue attempts on the code that lies beyond the conditional.

6.3 Improved Input Scoring

As a consequence of the large slowdown imposed on the execution, it becomes also more important to intelligently pick previously generated input files to symbolically execute. Each analyzed input that does not bring the execution closer to uncovered code is essentially wasted time.

Our current ranking algorithm for potential test cases is simple. This becomes noticeable in the results, where long stretches of test cases are executed leading to no new line coverage. A more advanced algorithm could analyze the program's control flow graph from previous executions and prioritize inputs based on how close their execution traces operate to uncovered branches. It would also be sensible to implement some of the techniques AFL uses.

7

Conclusion

In this thesis we explored the concept of using binary analysis techniques to automate program testing. We described symbolic and concolic execution and its application to the field, and designed and implemented a system that applies this to test case generation. Some of the work has lead to corrected errors in the original open source project, as outlined in Chapter 4. We compared our system to a state-of-the-art fuzzer, American Fuzzy Lop, and determined that our solution has merit.

The results show promise when compared to AFL. Our solution performs better in certain cases of difficult branch conditionals where AFL consistently gets stuck. It is however clear from the tests with larger programs that our approach requires significant more work. In particular, the exploration of loops needs to be sped up significantly in order to get close to the performance of advanced fuzzers. This becomes very visible from the differences in the graphs for coverage by time, and coverage by test cases. While the produced test cases are of similar or better quality than those of AFL, the speed of analyzing them in the current version is very low.

We conclude that symbolic execution is a promising field in dynamic program testing, and that we have contributed to the field by implementing a prototype using it on top of the promising PANDA framework. We do however realize that our prototype and the field requires more work in order to be competitive with more blunt program testing systems, and hope that this work will lead to future versions.

Bibliography

[Ait02] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 2002.

[BBGM12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 818–825, 2012.

[BEL75] Robert S Boyer, Bernard Elspas, and Karl N Levitt. SELECT&Mdash;a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[BMS11] Domagoj Babi, Lorenzo Martignoni, and Dawn Song. Statically-Directed Dynamic Automated Test Generation. *International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, pages 443–446, 2008.

[BSST09] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of satisfiability*, 185:825–885, 2009.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[CBP+13] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Laszlo Szekeres, Stephen McCamant, and Dawn Song. Transformation-aware Exploit Generation using a HI-CFG. 2013.

[CC10] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. 2010.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically generating inputs of death. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*. ACM, 12 2006.

[CGZC09] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *SIGPLAN Not.*, 47(4):265–278, 3 2011.

[Cla76] Lori A Clarke. A Program Testing System. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[CVE] National vulnerability database. `https://web.nvd.nist.gov/view/vuln/statistics-results?cves=on`. Accessed: 2015-05-25.

[DGHH14] BF Dolan-Gavitt, J Hodosh, and P Hulin. Repeatable Reverse Engineering for the Greater Good with PANDA. 2014.

[DGLHL13] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan Zee (north) bridge: mining memory accesses for introspection. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850, 2013.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[FM00] Justin E Forrester and Barton P Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, 2000.

[FPS11] Andrea Flexeder, Michael Petter, and Helmut Seidl. Side-effect analysis of assembly code. In *Static Analysis*, pages 77–94. Springer, 2011.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices*, 43:206, 2008.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, 6 2005.

[GLM+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated Whitebox Fuzz Testing. *NDSS*, 2008.

[God09] Patrice Godefroid. Software Model Checking Improving Security of a Billion Computers. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, page 1, Berlin, Heidelberg, 2009. Springer-Verlag.

[How77] William E Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 7 1977.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KKS98] Nathan P N.P. Kropp, P.J. Philip J Koopman, and D.P. Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239, 1998.

[Lat02] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[LBB+07] Timothy Robert Leek, Graham Z Baker, Ruben Edward Brown, Michael A Zhivich, Richard Lippman, and Richard P Lippmann. Coverage Maximization Using Dynamic Taint Tracing. Technical Report March, DTIC Document, 2007.

[MCM06] Barton P Miller, Gregory Cooksey, and Fredrick Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. 2006.

[MFS90] Barton P Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, 12 1990.

[MKL+95] Barton Paul Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[MPC+13] Stephen McCamant, Mathias Payer, Dan Caselden, Alex Bazhanyuk, and Dawn Song. Transformation-Aware Symbolic Execution for System Test Generation. *Univ. Calif. Berkeley*, 2013.

[NMRW02] George C Necula, Scott Mcpeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *International Conference on Compiler Construction*, pages 213–228, 2002.

[QR11] Xiao Qu and Brian Robinson. A Case Study of Concolic Testing Tools and their Limitations. *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 117–126, 2011.

[RHC76] Chittoor V Ramamoorthy, Siu-Bun F Ho, and W T Chen. On the automated generation of program test data. *Software Engineering, IEEE Transactions on*, (4):293–300, 1976.

[SAB10] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis Forward Symbolic Execution (but might have been afraid to ask). *Policy*, pages 1–5, 2010.

[SBY+08] 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. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5352 LNCS:1–25, 2008.

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Logics in Artificial Intelligence*, pages 308–319. Springer, 2002.

[Tur36] Am Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 38(1931):173–198, 1936.

[WLK13] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7791 LNCS:144–163, 2013.

[Zit03] Misha Zitser. *Securing software: An evaluation of static source code analyzers*. PhD thesis, Massachusetts Institute of Technology, 2003.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Software Engineering Notes*, 29(6):97, 2004.

[ZLL05] Michael Zhivich, Tim Leek, and R Lippmann. Dynamic Buffer Overflow Detection. *2005 Workshop on the Evaluation of Software Defect Detection Tools 2005*, pages 1–7, 2005.