NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
M.S. THESIS APPROVAL FORM

THESIS TITLE: Platform Agnostic Binary Coverage Maximization

AUTHOR: Francis V Adkins

M.S. Thesis Approved as an Elective towards the Master of Science Degree in Computer Science.

[Signatures and dates]

4/21/2015
Date

4/17/2015
Date

21 Apr 2015
Date

04/21/2015
Date

GRADUATE-SCHOOL APPROVAL:

[Signature]

Director, Graduate School

4/21/2015
Date

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[Signature]

Recipient’s Signature

4/21/2015
Date

Distribution: Once completed, this form should be scanned and attached to the front of the electronic Dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Platform Agnostic Binary Coverage Maximization

Author: Frank Adkins
Supervisor: William Robertson

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science

April 2015
Disclaimer

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.
"We cannot do everything at once,
but we can do something at once."

Calvin Coolidge
Software security hinges on the need for developers to find vulnerabilities in their software before malicious users do. However, input test suites are manually intensive to produce and often fail to maximize coverage. This generally prescribes the use of a fuzzer. Yet, modern techniques also fail either because they lack the information necessary to make progress or require too much to be practical. In our research, we aim to strike an ideal balance in this spectrum and make an extra effort to address architectures that are commonly excluded from security research.

In this thesis, we approach the concept of automated test case generation as it pertains to user-controlled input files. Our primary targets are those which consume structured input such as media or compression parsing libraries. Given only a stripped binary, our goal is to maximize the coverage of that binary by generating a thorough test suite. Utilizing the PANDA platform, we employ taint tracing and branch coverage techniques to drive execution to previously uncovered code. With these methods, we demonstrate significant increases in code coverage in certain cases and highlight the potential for immediate practical application. Furthermore, we demonstrate prioritization techniques to quickly enhance branch coverage and also describe a novel method to retrieve otherwise inextricable information by composing analyses of a target compiled for multiple architectures or at multiple optimization levels.
Acknowledgements

I would first like to extend a heartfelt thank you to Tim Leek, Ryan Whelan, and Patrick Hulin for all of their help with PANDA. This could not have happened without their support.

I would also like to thank Kyle Ingols and Group 102 at MIT Lincoln Laboratory. You gave me the opportunity to be here today, for which I am humbled and grateful.

Finally, a special thank you to Dr. Robertson and Dr. Kirda, my advisors at Northeastern, whose insight and knowledge guided me through this process.
# Contents

Disclaimer ........................................... i
Abstract ................................................ iii
Acknowledgements ....................................... iv
Contents .................................................. v

## List of Figures

List of Tables ........................................... viii

## Abbreviations

1 Introduction ........................................ 1
   1.1 Introduction ..................................... 1
   1.2 Motivation ......................................... 1
   1.3 Test Case Generation ............................... 3
   1.4 PANDA .............................................. 4
   1.5 Approach ........................................... 5
   1.6 Contributions .................................... 6

2 Related Work ......................................... 7
   2.1 Coverage Maximization and Fuzzing ................. 7
      2.1.1 Fuzzing ....................................... 8
      2.1.2 Dynamic Taint Analysis ....................... 9
      2.1.3 Symbolic Execution ............................ 11
      2.1.4 Miscellaneous ................................ 12
   2.2 Automatic Exploit Generation ....................... 13
   2.3 Summary ............................................ 14

3 System Overview ...................................... 15
   3.1 Introduction ...................................... 15
   3.2 How PANDA Works ................................... 15
      3.2.1 QEMU .......................................... 16
      3.2.2 LLVM ........................................... 16
      3.2.3 JIT Translation ................................ 17
   3.3 Record and Replay .................................. 18
   3.4 Plugins ............................................ 18
      3.4.1 OS Introspection ............................... 19
      3.4.2 Taint System ................................... 19
      3.4.3 Dynamic Values ................................. 20
List of Figures

1.1 CVEs Reported by Year .................................................. 3
2.1 The Coverage Frontier ..................................................... 8
3.1 Classical Compiler Architecture ........................................ 16
3.2 LLVM Architecture .......................................................... 17
3.3 Coverage Maximization Architecture ..................................... 21
3.4 Picking Prioritization ....................................................... 23
4.1 Annotated Coverage Maximization Architecture ......................... 26
4.2 Recording Process ............................................................ 29
4.3 Identical LLVM IR Across Two Runs ..................................... 31
5.1 Static Value test results ..................................................... 37
5.2 S1 coverage test results ..................................................... 41
List of Tables

5.1 Average x86 execution time for static value tests .......................... 39
5.2 Average stage execution time for full i386 s1 testing ...................... 42
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>American Fuzzy Lop</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>CIL</td>
<td>C Intermediate Language</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>LLVM</td>
<td>(Formerly) Low Level Virtual Machine</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>TCG</td>
<td>Tiny Code Generator</td>
</tr>
<tr>
<td>TCN</td>
<td>Taint Compute Number</td>
</tr>
</tbody>
</table>
To my loving wife,
who put up with the sound of my mechanical keyboard.
Chapter 1

Introduction

1.1 Introduction

It is a well accepted mantra of the software security community that, aside from formally-proven systems, no program is completely secure. Yet, across the spectrum of software vulnerabilities, from memory corruption bugs to cryptographic errors, there is one consistent factor: the vulnerability was written by a developer. In an ideal world, it would be impossible to introduce bugs and all code would perfectly reflect the developer’s intention with no room for ambiguity or unintended consequences. However, in reality, the only hope a developer has at redemption is through an exhaustive test suite.

In this thesis, we approach the concept of automated test case generation as it pertains to user-controlled input test files. The primary targets of interest are those which consume structured input such as image/media parsing libraries, PDF readers, compression parsers, or similar subjects. However, the subjects need not necessarily be file-consuming as the approach can also be extended to network-based services with simple test harness adaptations.

1.2 Motivation

The basic concept of a software bug lies in the existence of some input that causes the program to react in an unexpected and non-ideal way. Occasionally, this input can have security implications for the system overall. In the best case, this may cause a program crash, which, alone, could have serious ramifications if the nature of the program requires 100% up-time. This is often true of medical, avionic, or other safety-critical applications. In the worst case, such a bug has the potential to be intentionally exploited by a malicious
user to gain control of the program’s execution for their own nefarious ends. In either extreme, such a bug is commonly defined as a software vulnerability.

To mitigate the number of vulnerabilities that may be present in a program, it is common practice to develop suites of test input files that can be processed and monitored for proper execution. In many cases, however, this suite fails to exercise the full functionality of the program under test. This can leave glaring errors with the potential to yield devastating results during execution. It is therefore reasonable to expect a thorough and robust test suite as a prerequisite for any software product with claims to either robustness or security. However, the act of achieving 100% code coverage with these tests can often be non-trivial for a number of reasons:

First, producing test cases is generally a manual process and requires significant domain knowledge. Ideally, these tests should be created by the developer who initially wrote the code as well. This manual process tends to be monotonous, onerous, and highly error prone [1, 2]. Additionally, in more complex projects, it can be next-to-impossible to conceptualize a test that would exercise a specific portion of the code. In such a case, the Pareto principle may come into effect as the developer reaches a “good enough” testing base, potentially failing to exercise a small yet critically-important portion of their code.

Second, for a variety of reasons, it has become common practice to apply combinations of programming languages to bring about the final product. This also increases the difficulty for test authors to reach 100% code coverage as test cases traverse boundaries between components. This prompts a number of complications, not the least of which is understanding how to record coverage metrics at all.

Whatever the case, vulnerabilities continue to permeate the common code bases we use on a daily basis. As a testament to this, 2014 alone heralded a number of high-profile vulnerabilities (and their mass-media personifications) in major open-source libraries, including the HeartBleed, ShellShock, and GHOST vulnerabilities. Additionally, despite advances in static verification techniques and other beneficial code analysis capabilities, there appears to be no noticeable decline in the number of vulnerabilities reported each year [3]. This is demonstrated in Figure 1.1 which visualizes the number of Common Vulnerabilities or Exposures (CVEs) reported yearly.

While static verification and other tools play their part, it is still logical to expect developers to produce expansive enough test suites to support some claims of robustness and to mitigate any glaring issues that occur during ongoing development. However, the issues described above present a significant hurdle and beg for an automated approach.
This demand has subsequently spurred extensive research into automated test-case generation and the related subfields of fuzzing and automated exploit generation [1, 2, 4–19].

### 1.3 Test Case Generation

The primary impetus in generating test input files is to exercise the entire functionality of a program and ensure its proper execution under every possible scenario. The automation of this process can span the entire spectrum of complexity from simple random-input fuzzing to highly intricate analyses of the program under test. These approaches also differ in the quantity and quality of information provided about the target program to assist in developing new test inputs.

Within this concept, there are three general categories which describe the amount of information given to an automatic input generator: white-box testing, black-box testing, and grey-box testing [2, 20].

White-box testing assumes that the input generator has full access to the source code of the program under test and even has the ability to recompile it as necessary to include additional instrumentation.
Black-box testing assumes that the source code for the program is not available. It also assumes that any internal state introspection is unavailable. The only information that can be gleaned by these systems is via monitoring externally-visible behavior such as return codes and running status.

Grey-box testing is a hybrid of these approaches. Like black-box testing, it assumes that source code is unavailable. However, unlike black-box approaches, internal state information is retrievable via some form of dynamic program instrumentation. This effectively allows the system to retrieve additional information from the binary as it runs, even if it is stripped of debugging symbols.

In this thesis, we take a grey-box approach to prevent the necessity for source code. This offers a range of benefits which are delved into more fully in subsequent chapters.

As a note on terminology: while there is some dispute on the definitions of these terms, we opt for the those provided in [2] and [20]. Other sources would contend that any form of program introspection should be considered white-box analysis. However, we believe that this label is too ambiguous and fails to differentiate analyses which do and do not require source code. To emphasize our non-reliance on source, we opt to use the term grey-box to describe our approach.

To conduct our research, we rely on a nascent dynamic analysis platform called PANDA.

1.4 PANDA

PANDA is the Platform for Architecture-Neutral Dynamic Analysis. It is currently under development by members from MIT Lincoln Laboratory, Georgia Institute of Technology, and Northeastern University. It has also been the subject of a few recent notable publications [21–23].

PANDA is powered by the composition of two potent technologies: the QEMU emulation system, and the LLVM compiler infrastructure project.

QEMU is an open-source project that emulates a variety of instruction set architectures (ISAs) for the x86 processor [24]. For instance, with QEMU, it is possible to emulate either a single binary or entire operating system compiled for ARM, MIPS, PPC, or a number of others on an x86 platform.

The LLVM compiler infrastructure project is a straightforward rethinking of how modern compilers should work. It operates on the concept of decoupling the compilation stages and simplifies the additional of new languages or targets. It also offers a powerful set of
libraries to manipulate the LLVM Intermediate Representation (IR) which serves as an intermediary between the input source code and the binary product.

PANDA has combined these two technologies by modifying the QEMU system to use the LLVM IR as its execution medium. This power comes from its ability to perform Just-In-Time (JIT) translation of the QEMU Intermediate Representation, known as the Tiny-Code-Generator (TCG) IR, into the intermediate representation used by the LLVM compiler framework (LLVM IR). It is then possible to administer highly powerful IR analyses provided by the LLVM framework as the code executes.

To assist analysts further, the PANDA platform also provides a simple plugin interface and the ability to communicate with existing plugins such as a sophisticated taint-tracing system [21–23]. This supports the ability to trace the exact influence that an input file (or other region of memory) has on various parts of the execution of the target binary or system.

For additional information on PANDA and its capabilities, please see subsequent chapters in which we delve more fully into the features it provides. The versatility of this tool is also demonstrated in their publications as well as in various examples on GitHub [21–23, 25].

1.5 Approach

In this thesis, we approach the issue of automatic test file generation assuming only the presence of a stripped binary executable and, while not required, a potential corpus of input files. Given this initial corpus, we conduct grey-box dynamic analysis on the program as it consumes each input file in turn. We can then attempt to derive useful information for future test candidates to iterate the process and achieve 100% coverage of the binary. Specifically, we utilize the PANDA taint-tracing system to identify key branch conditions within the target program which may be subject to influence by the current input file.

From this taint-tracing information, we can identify precisely which bytes of the input file influence key branch conditions and can conduct additional analyses of the LLVM IR at that location to intelligently mutate the original test file. After this process completes, the mutated child candidates are added to the initial corpus for another iteration and another attempt to broaden overall coverage.
1.6 Contributions

The contributions of this work are:

- A platform to conduct grey-box dynamic analysis of a program and identify specific input bytes which influence branch conditions within that program.

- A method to measure 100% “true” instruction coverage, rather than 100% source coverage to unmask compiler introduced discrepancies.

- Description of a metric to determine when test file mutation may reasonably halt.

- Description of a scoring method to prioritize input files most likely to yield increased coverage.

- Description of a method to compose analyses of a target program compiled for multiple architectures or optimization levels to derive otherwise inextricable information.
Chapter 2

Related Work

This work relies on a wide corpus of previous research. Here, we summarize some relevant advancements in the fields of Coverage Maximization, Fuzzing, and Automatic Exploit Generation.

2.1 Coverage Maximization and Fuzzing

Typically, one of the primary goals of any automatic test file generation system is to maximize the amount of executable code covered by the test corpus. Some systems, including fuzzers and automatic exploit generators, seek the additional goal of uncovering a vulnerability to exploit. However, since many of these vulnerabilities lie in the seldom-tested fringes of a code base, it is well-acknowledged that code coverage maximization is a good place to start.

In their COMET research, Tim Leek et al. support the claim that code coverage is maximized whenever both the true and false branches of every reachable conditional statement in a program have been executed [10]. To represent this, they provide a model of the coverage frontier, shown in Figure 2.1.

Given the consolidated coverage achieved thus far by a corpus of input files, the coverage frontier can be defined as “the set of conditionals for which only one of the two possible branches was ever taken” [10]. Thus, the end goal is to target the remaining, or half-covered, conditionals. Code coverage is then increased whenever execution can be driven to reach the branch destination that has not yet been covered. This or similar models have been accepted in a variety of publications and have yielded numerous approaches.
2.1.1 Fuzzing

The term *fuzzing* applies to the broad goal of finding an input which is able to uncover a vulnerability within a target program [26]. In this section, we consider some black-box approaches to fuzzing and identify their pros and cons.

The most basic form of fuzzing is random generation, in which a purely random file is constructed and fed into the target. In the early days of fuzzing, this approach met with significant success [14, 26–30]. However, as fuzzing became more commonplace, product developers began to exercise such testing themselves and minimized the number of vulnerabilities that could be found externally in this way.

Additionally, this approach suffers from the major limitation that it often fails to bypass initial file format checking. This results in very shallow program coverage as each input file is generally rejected before progressing to the program’s inner logic [9, 10].

To address this hurdle, the approach was modified to instead randomize a subset of bytes within an existing valid file. This results in tests that are able to bypass initial file format checks and fuzz more deeply within the target program’s logic. This technique is still commonly applied today and can be found in tools such as Zzuf [31].

The benefits of such an approach are that it requires almost no knowledge to employ and can conduct a large number of tests in a short amount of time due to the relatively small amount of computation needed. The downsides are that it can fail to bypass what amount to ultimately simple checks, such as a test of: if \((x == 10)\) [7]. This is primarily due to the black-box nature of the approach, which has no introspection capabilities and is only able to make progress via random mutation. Therefore, techniques such as
Chapter 2. Related Work

this can take a significant amount of time to randomly generate a valid input and make further progress.

Additionally, this approach requires that a corpus of valid input files already exists. For any breadth of coverage, these input files must also exercise large portions of the target program’s functionality and features. Tests such as these may not always be available, especially in cases involving proprietary systems.

To combat these simple limitations, the next evolution of black-box fuzzers were grammar or model based. They utilize the construction of a grammar which specifies the precise format of an acceptable input file. This category encompasses some of the most common commercial fuzzers in use today including both Peach and Sulley [32, 33]. Both require the manual construction of an input grammar from which random permutations can be produced. Obviously, the primary limiting factor in this approach is the manual effort required to construct these grammars, which can be significant for more complex format specifications. This manual limitation has subsequently hindered the uptake of such fuzzers.

To address this, some effort has also been made to automatically generate the necessary grammars. An example of this can be found with [34]. Their method involved inferring data structures from a preexisting corpus of valid input files. From these inferred structures, they could then automatically generate acceptable grammars and therefore produce fuzzed output. While their initial analyses proved intractable, subsequent work yielded fairly accurate grammars and helped uncover four vulnerabilities in the five antivirus programs tested. Automated methods such as this have their merits and are viable candidates for future research.

In practice, these systems perform well at uncovering vulnerabilities and yield promising results. However, their black-box nature relies on unguided randomness to make progress and generally limits their efficacy at maximizing code coverage. Our work moves beyond these limitations and acknowledges the need for more information to make appreciable progress.

2.1.2 Dynamic Taint Analysis

Adapting a more complex introspection model enables the use of powerful techniques such as dynamic taint analysis. Dynamic taint analysis involves tracing relevant values (such as user input) throughout a program’s execution and identifying other values that are influenced or tainted by them.
This can have a variety of potential applications. For example, Polyglot was a system designed by Caballero et al. that utilized a custom version of QEMU to monitor a program as it consumed network packets. They marked these packets as tainted and analyzed how the individual bytes were processed within the target. Using this technique, they were able to designate groupings of bytes and ultimately reverse engineer the protocol format with significant accuracy \[35\].

However, for the purposes of test case generation, dynamic taint analysis is especially relevant in tracking which branch conditions can be affected by the bytes of the input file. From this information, we can determine which branches on the coverage frontier can be targeted directly to maximize coverage.

Generally, taint analysis requires greater introspection abilities than those provided by previous black-box strategies. This usually entails either a white-box or grey-box approach to implement program instrumentation. Such instrumentation inserts additional behavior during the program’s execution and can be used to incorporate code tracing, debugging, profiling, or other metrics that would not otherwise be available via static analysis. This approach is also often used to implement dynamic taint analysis \[36\].

For example, the COMET and subsequent BuzzFuzz systems (upon whose concepts this thesis is based) utilized a white-box instrumentation technique via a C Intermediate Language (CIL) \[37\] source-to-source transformation \[9, 10\]. This allowed them to include additional behavior to implement taint analysis and also identify whether the true or false blocks were taken on each conditional branch in a C program. From this, they could determine both which branches were taken and derive the specific input extents (bytes) affecting those branches.

Using this method, they achieved promising results and uncovered a number of bugs in their targeted libraries. However, this did have several pitfalls, including the necessity of source code. Specifically, the need for source code required them to instrument every library used by their target program to maintain taint flow across boundaries. This also prohibited the analysis of taint flow throughout the kernel or between processes. This ultimately constrained their operating space to an extent that might not be practical in real-world operation.

Our work seeks to achieve similar coverage and vulnerability discovery results, yet mitigate the limitations inherent in requiring source code by instead opting for a whole-system emulation approach to taint analysis.
Chapter 2. Related Work

2.1.3 Symbolic Execution

Symbolic execution is a topic that has been covered numerous times in recent literature, so we will only provide a cursory summary of its benefits as they relate to test case generation here.

Symbolic execution is a term used to describe the symbolic interpretation of variables within a program throughout the path of execution. In other words, it involves interpreting a variable as an unknown rather than as some concrete value and aggregating the conditions that the variable must conform to to reach a given point in the program’s execution. During this symbolic interpretation, it is then possible for a tool to construct a list of the path constraints encountered by each variable and solve them to produce a candidate concrete value.

As it relates to test case generation, it is common practice to invalidate one of the constraints imposed by a branch condition and utilize a solver to generate a model for a test file that would have the opposite result at that branch condition. Ultimately, in our terminology, this approach is used to address a specific half-covered conditional and can use a solver to yield a file that is able to drive toward previously uncovered code.

An approach such as this was utilized by the CREST tool in [38]. In this case, the team utilized the increasingly popular strategy of concolic execution - a process through which a concrete test input is processed simultaneously with symbolic execution. This simplifies the constraints that are generated and creates a more scalable approach than symbolic execution alone. In CREST, they used a CIL source-to-source transformation which allowed them to extract the control-flow graphs of the program under test. This, coupled with the Yices solver [39], allowed them to generate much better coverage than random testing.

The SmartFuzz tool by Molnar et al. utilized a similar approach, yet directed their efforts specifically at uncovering integer bugs such as overflow/underflow, width conversions, and signed/unsigned conversions [40]. They simplified analyses by utilizing the VEX intermediate representation provided by Valgrind [41] and instrumented at the IR level to provide both symbolic constraint generation and dynamic taint analysis. They found that by storing symbolic information strictly for tainted memory, they were able to greatly reduce the size of constraint systems and also reduce the memory overhead of symbolic execution.

Another popular tool in this category is the SAGE system, developed by Michael Levin, David Molnar, Patrice Godefroid, and others [7, 40, 42, 43]. SAGE is directed at generating test files for Windows x86 systems and differs from many previous attempts in that
it adopts a machine-code-based approach to symbolic execution rather than relying on
source code. SAGE is well acknowledged as a success in the field of symbolic execution
and has revealed a number of bugs in thoroughly tested Windows code. This success
is largely due to a significant number of optimizations including off-line program execution
for constraint generation. SAGE is closed-source and is now a Microsoft research
product.

A final heavyweight contender in this arena is the KLEE tool developed by Cadar et
al. [8]. KLEE conducts symbolic execution by instrumenting code at the LLVM Inter-
mediate Representation level and employs a large number of optimizations in constraint
solving and program state representation to achieve reasonable execution time. Accord-
ing to their documentation, KLEE has two goals: (1) hit every line of executable code
in the program and (2) detect at each dangerous operation (e.g., dereference, assertion)
if any input value exists that could cause an error. They boast impressive coverage
metrics and have previously uncovered a number of serious bugs, including some in the
heavily-tested GNU COREUTILS suite.

The KLEE project is particularly significant to our research because of its use of the
LLVM Intermediate Representation. PANDA also utilizes LLVM IR and thus highlights
a promising direction for future research. See Section 6.1 for more on this.

For more information on dynamic taint analysis, symbolic execution, and the com-
putational hurdles therein, please see All You ever Wanted to Know About Dynamic
Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask) by
Schwartz et al. [44].

2.1.4 Miscellaneous

In addition to the general approaches of taint analysis and symbolic execution, several
other sophisticated methods have been applied to coverage maximization.

For example, a unique approach was taken by Sparks et al. who analyzed the control
flow graph (CFG) of a binary [45]. After picking a specific target node in the CFG, they
were able to represent the graph as an absorbing Markov chain and specify weights in
the graph transitions. This model served to yield fitness ratings for each input, which
could then be used to derive new inputs to drive toward the target node. This approach
was able to proceed deeper into the CFG in much less time than achieved by random
fuzzing.

Next, American Fuzzy Lop (AFL) is a fuzzer developed by Michal Zalewski [46] that
has “street-smarts”. It implements a form of lightweight instrumentation at either a
white-box (via custom g++ and LLVM compilers) or grey-box (via a modified QEMU user-mode emulator) level strictly to measure transitions between basic blocks. It then utilizes custom mutations on an input file and favors those inputs which generate new block transitions. In this way, it steadily increases coverage by emphasizing tests which traverse the code in new ways. In practical application this technique has been highly successful and has identified a significant number of vulnerabilities in common libraries.

AFL is interesting for our research because it provides a lightweight means to quickly generate significant coverage. It operates on an initial corpus of input files and produces a more broadly-scoped final set of files. In this way, it is highly adapted to supplement more heavyweight analyses such as our own and could yield even greater results if effectively coupled. See Section 6.1 for more on this.

### 2.2 Automatic Exploit Generation

Finally, we would be remiss to not mention the significant work being produced in automatic exploit generation. This field is strongly aligned with the goals presented above, yet takes the additional effort to produce working exploits for relevant security vulnerabilities.

In his thesis, Sean Heelan constructed a novel system that integrates taint propagation and constraint solving to automatically generate a control flow hijacking exploit when given only a target program and an input that causes it to crash [36]. This could be a great supplement when coupled with a fuzzer capable if producing such an input.

Finally, there is one heavyweight contender in field of automatic exploit generation and that is the research produced at Carnegie Mellon under David Brumley et al. [5, 6, 47]. Their most recent research is entitled: Unleashing Mayhem on Binary Code. This paper describes an end-to-end system that is able to produce working exploits given only a stripped binary executable. The Mayhem system utilizes symbolic/concolic execution as well as taint analysis to generate a crashing input. From this input it then generates a full control-flow hijacking exploit.

Mayhem provides numerous mitigations to attempt to overcome the traditional hurdles of symbolic execution and has demonstrated phenomenal results in vulnerability discovery. However, computation complexity is still very much an issue and more work is needed in these fields.
2.3 Summary

In reviewing the work described above, we reached several conclusions which motivated this research.

First, black-box systems are approaching the limits of what they can accomplish regarding coverage maximization. For any hope at achieving additional coverage (which is not entirely dictated by random chance) more information about the program under test is necessary.

Second, the requirement of source code to conduct analyses is a major limitation for practical systems. It places boundaries on where analysis can take place, requires significant effort to support intersections between languages, and is of no use when testing proprietary or closed-source programs.

Third, many, if not all, of the systems above are strictly targeted to the x86 architecture. This leaves significant ground uncovered, particularly for the mobile community.

Finally, while symbolic execution is promising, it continues to face issues with intractability. Additionally, major competitors in the field tend to be closed-source and limit any incremental improvements that could otherwise be achieved via open source collaboration.

To address each of these issues, we have identified the PANDA platform as the ideal system for our analyses.

- It provides a standard analysis medium in the form of LLVM IR to mitigate programming language differences and has no requirement of source code.
- It supports a large number of architectures, including ARM with support for Android.
- It provides a robust and tractable taint-tracing system for practical application.
- It facilitates an easy-to-use plugin interface to provide powerful analyses.
Chapter 3

System Overview

3.1 Introduction

Our system is built using the highly versatile PANDA platform. As mentioned in Chapter 1, PANDA is the Platform for Architecture-Neutral Dynamic Analysis [21–23, 25]. It provides robust grey-box analysis capabilities via an easy-to-use plugin interface. From this interface, it is possible to insert custom instrumentation at precise locations during the system’s execution. This facilitates the ability to create analyses that are powerful yet still tractable.

PANDA is highly versatile and has been applied to several reverse engineering and program analysis tasks. These have included reverse engineering the CD key algorithm of the original Starcraft, retrieving the master secret from an SSL session to decrypt network traffic, and uncovering the censorship activities of a Chinese IM client, just to name a few [22, 23].

3.2 How PANDA Works

PANDA operates as a modified version of the open-source QEMU emulator. The current version supports QEMU’s whole-system mode which provides emulation of a full operating system for a given Instruction Set Architecture (ISA). Its primary power comes from its ability to conduct sophisticated analyses on instructions as they execute. This power is ultimately a result of the intersection between the QEMU and LLVM projects.
Chapter 3. System Overview

3.2.1 QEMU

QEMU is an emulation platform that allows targets compiled for a variety of platforms to run on an x86 ISA [24]. For example, QEMU can be used to emulate either an individual executable or entire operating system compiled for the ARM ISA to successfully execute on a traditional x86 processor.

To provide support for such a wide variety of ISAs, QEMU uses a standardized instruction Intermediate Representation (IR) known as the Tiny Code Generator (TCG) IR. This simplifies the process of supporting additional ISAs in that it is only necessary to create a new front-end to translate the guest ISA to TCG IR. The TCG IR is then able to execute independently on the host processor. In this way, QEMU has added support for more than 10 ISAs.

However, there are a relative dearth of tools to manipulate TCG IR, which means that it is often not the most efficient medium through which to conduct complex instruction-level analyses. Therefore, to provide more power, we turn to the LLVM compiler infrastructure.

3.2.2 LLVM

LLVM is a compiler infrastructure project that has greatly risen in popularity since its inception [48]. Its primary goal is to vastly simplify the architecture used in previous compiler implementations such as GCC [49]. To do so, it, like QEMU, provides independent front and back ends. LLVM splits each stage of the classical three-phase compiler (as seen in Figure 3.1) into its own atomic component.

**Figure 3.1: Classical Compiler Architecture**

![Diagram excerpted from [50]](image)

In a classical compiler architecture it has generally necessary for each part of the three stage process to be able to communicate with one another. For example, to produce debugging symbols, the back-end may need to consult with the front end to access portions of the source code. This eventually yields a system in which the addition of support for a new architecture requires intricate knowledge of all parts of the compiler and how they interact.
Conversely, each stage in the LLVM architecture is entirely independent and requires a developer only to be familiar with the specific portion they are working on. Thus, it is possible for a front-end to independently translate each source language directly to LLVM IR. From there, optimization takes place strictly at the IR level with no requirement for the original source. Finally, the back-end translation can subsequently convert the resulting LLVM IR to the target ISA. This segmented architecture is depicted in Figure 3.2.

Figure 3.2: LLVM Architecture

Diagram excerpted from [50]

As it pertains to PANDA, the middle optimization step is particularly interesting. The LLVM platform provides a set of powerful libraries to assist in manipulating code at the LLVM IR level. These libraries are capable of acting anywhere from the full-program level down to individual instructions.

3.2.3 JIT Translation

To achieve the power needed for its complex analyses, PANDA utilizes a TCG to LLVM IR translation during runtime as adapted from the S\textsuperscript{2}E project [51]. It is then possible to instrument the LLVM IR as needed and run the result using the LLVM Just-In-Time (JIT) engine. This completely replaces the TCG IR execution engine typically used by QEMU.

This obviously has a performance impact and is not intended for full-time operation. Instead, LLVM translation can be toggled on only when necessary. In all other cases, TCG execution continues as normal. Many complex analyses such as the taint functionality rely on this capability to achieve reasonable execution times. Subsequently, it can be signaled to only begin LLVM translation when requested from the guest. Other computational issues arising from LLVM translation are mitigated via the offline analysis capabilities provided by PANDA’s record and replay functionality.
3.3 Record and Replay

One of PANDA's most powerful features is its ability to support whole-system deterministic record and replay \[25\]. This is facilitated by the snapshotting capabilities that are provided by the QCOW image format. To conduct a recording, PANDA keeps a log of all non-deterministic inputs provided to the system. At a high level, these can be thought of as things like network packets, hard drive reads, and mouse/keyboard inputs. At a more detailed level, these include all cases in which changes are externally made to either the CPU state or memory, such as Direct Memory Access (DMA), interrupts, or x86 “in” instructions.

This non-determinism log, coupled with a QCOW external snapshot of the current state of the system, provides enough information to deterministically recreate the entirety of a recording. Note however, that PANDA does not track the state of each external device and therefore cannot be used to “go-live” from a recording \[23\]. Instead, PANDA's recordings are simple and relatively compact. For example, experiments with recording the entire FreeBSD boot process consumed only 533MB on disk \[23\]. This is small given that the full recording was 9.3 billion instructions in length.

Ultimately, PANDA’s record and replay functionality is ideal for deterministically conducting analyses at increasing levels of complexity or conducting analyses that would not otherwise have been possible during active runs. For example, PANDA provides utilities such as the Scissors plugin which can be used to trim an existing recording down to the specific instructions of interest. This can vastly narrow the range of instructions to process and greatly assists in building tractable systems. Additionally, in some cases, plugins such as the taint-tracing system are simply too slow to be used during active runs. For instance, some network-based services may time-out while waiting for a response \[23\].

A recording allows complex analyses to have as much time as they need without the worry of corrupted or interrupted tests. Given a non-determinism log and a snapshot, the user can simply replay it with whichever plugins are desired at the time while maintaining an identical environment on each run.

3.4 Plugins

PANDA provides an easy-to-use plugin interface which supports numerous locations to register dynamic callbacks.
As a basic example of this, one of the most common callback locations is accessed via the PANDA_CB_BEFORE_BLOCK_EXEC definition. By registering a callback for this location, a function can be called prior to the execution each translated basic block. The arguments to this callback include the current environment of the emulated processor as well as the contents of the translated block itself to be queried or manipulated as desired. For more information on potential callback locations and capabilities, see Appendix B.

It is also possible for plugins to communicate amongst themselves to integrate various sources of information into a more expansive picture. This has the added benefit of promoting separation of duties and prevents unnecessary duplication of code. We have adopted this approach in our research. Some of the more relevant plugins we have used are detailed below.

### 3.4.1 OS Introspection

PANDA plugins inherently execute at a very low level. Specifically, they are only given a picture of the guest that is seen by the hardware itself. Thus, it can be difficult to achieve a higher-level understanding of what is happening in the Operating System (OS), such as which process is currently executing. To combat this, convenience plugins for several operating systems have been implemented which can be queried for this information from other plugins.

This is particularly useful in our research to differentiate between branches encountered in the target program from those encountered in other processes running at the same time.

### 3.4.2 Taint System

The PANDA taint system makes extensive use of the LLVM instrumentation libraries and implements taint-tracing at the LLVM IR instruction level. This has a major benefit of being applicable across all architectures supported by PANDA, including both x86 and ARM.

Because LLVM execution is so computationally demanding, it is only enabled at the exact moment that it is required. Specifically, this is enabled via a *hypercall* instruction from the emulated guest under analysis. This *hypercall* is dependent on the guest architecture but is generally an infrequently used instruction with additional free registers such as the `cpuid` instruction on x86. Note that PANDA does not modify the kernel of the guest OS and thus this *hypercall* terminology used here strictly refers to the message passing instruction and not full Paravirtualization.
Once a hypercall is detected with the appropriate arguments, the given memory region is marked as tainted and is then tracked via a shadow memory region for the duration of the recording. The whole-system approach to emulation grants PANDA a great amount of power and allows it to trace taint across the hard drive, over the network, and throughout the kernel. This effectively mitigates the issues faced by previous white-box approaches and allows it to be applied to a variety of programming languages, operating systems, and ISAs.

As an added benefit, it is also possible to register a callback function with the taint plugin that will be triggered whenever it detects that the condition of a branch instruction has been tainted. This on_branch callback provides the exact LLVM register number upon which a branch was based as well as the taint labels placed on it and the register’s Taint Compute Number (TCN).

Taint labels, in this context, refer to a user-customizable set of labels that can be assigned to each byte of the tainted input. For our purposes these are simply the integer indices of the bytes of an input file from 0..n. Thus, the taint labels on a register form a set of integers which indicate from which input extents that taint was derived.

A TCN represents the number of operations that have taken place to yield the current tainted register from its initial state with TCN = 1. On each compute operation (addition, multiplication, etc.) the TCN is incremented. This information can then be used to determine which branch conditions are highly or only slightly correlated to their original input value. This provides us with nuance in selecting the next extents to be targeted and allows us to score which are the most likely to be the most productive for prioritization purposes.

For more information on the capabilities of the taint-tracing system, please see the original publication or documentation on GitHub [21, 25].

3.4.3 Dynamic Values

The final plugin used by our approach is the dynamic values or dynvals plugin. The dynvals plugin operates by instrumenting each LLVM block to include custom logging functionality. This functionality is then called during the course of that block’s execution. This can be used to record values which are only retrievable at runtime, such as the arguments to load, store, call, and branch instructions.
For our research, the values logged for branch conditions are particularly interesting. From this information, we can determine which branch was actually taken at each conditional. This ability allows us to identify the overall coverage frontier and, further, to isolate relevant half-covered conditionals.

### 3.5 System Architecture

Now that we have described the tools provided by PANDA, we can finally present a full architectural overview of our algorithm to achieve coverage maximization. A diagram of our approach is presented in Figure 3.3.

**Figure 3.3: Coverage Maximization Architecture**

3.5.1 Initialization

The intent in this research is to take an existing corpus of input files and mutate that corpus to increase the overall coverage of the target. Therefore, we begin our algorithm with some pre-arranged corpus of input files. This should ideally be a test-suite that exercises some breadth of the target’s functionality, but, in the worst case, may also be a collection of completely random files.
3.5.2 Find Tainted Branches

The primary mechanism that enables our algorithm is the process through which we determine which branch conditions can be influenced by the input file. As an added benefit, this step can also identify which specific bytes, or extents, of the input control each of these branches. This process is ultimately done by first fetching a test file from the input corpus, then running it against the target, and finally applying taint analysis to track the flow of data throughout the target’s execution. Each iteration of this process contributes to our overall picture of the coverage frontier.

3.5.3 The Coverage Frontier

Once we have gathered the set of tainted branches and the extents that influence them, we can merge this information into an aggregate database of total coverage. This total coverage database can further be filtered to identify the coverage frontier. This part of the algorithm is used to identify any remaining half-covered conditionals and will signal success if every influence-able branch has been covered. This stage also forms the basis for a novel \textit{number_attempts} counter.

As information from the previous stage is merged, we identify the addition of any branches that have been detected previously. If we find such a branch, we increment a \textit{number_attempts} counter on it. For instance, if two test files both encounter the same branch during their respective runs, then this branch will be incremented to have a \textit{number_attempts} value of 2. This counter provides two major benefits.

First, it forms the basis for a novel halting metric that is infrequently provided by most fuzzing frameworks. For example, with black-box fuzzers, it is often difficult to identify a reasonable point at which to stop fuzzing. The Microsoft SDL guidelines include an arbitrary metric of a minimum of 500,000 random iterations or at least 250,000 iterations since the last bug was found \[53\]. This milestone is put in place regardless of the effectiveness or results of the fuzzing itself.

Instead, the \textit{number_attempts} metric places a much more reasonable value of \(n\) attempts at each branch condition. This ensures that testing is much more well-rounded and targets all parts of the application to at least a reasonable minimum extent. This metric can also highlight any branches that are particularly difficult to exercise and would be fruitful candidates for manual analysis.

Second, the \textit{number_attempts} value also helps score which branches should be targeted next to yield the greatest possible results in the least amount of time. This prioritization is done during the picking stage.
3.5.4 Picking

The picking stage implements scoring in two primary steps:

First, it filters all half-covered conditionals remaining in the coverage frontier by those that are directly reachable from the current input file. This capability is entirely owed to our use of the taint system. This means that we only mutate files which can actually influence the coverage frontier and dismiss any that do not.

Next, this picking process prioritizes the remaining half-covered conditionals by sorting them according to the \textit{number_attempts} they have received thus far. Branches that have encountered fewer attempts receive a higher precedence and are targeted first. An example of this process is presented in Figure 3.4.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{picking_prioritization.png}
\caption{Picking Prioritization}
\end{figure}

In this example, the \textit{coverage frontier} has already been filtered to four candidate half-covered conditionals. The picking process may then be instructed to select two of these conditionals to be targeted during the next iteration of the algorithm. With these instructions, it would seek the two candidates with the smallest \textit{number_attempts} and subsequently select Conditional D, and Conditional A. In this way, the information \{\texttt{Branch:D, Extents:12}\}, and \{\texttt{Branch:A, Extents:1, 2}\} would be passed to the child generation stage.

3.5.5 Generate Children

The child generation stage takes the information provided by the picking process and seeks to intelligently craft a child input that will drive coverage down the previously uncovered portion of the conditional. To do so, it can employ any number of techniques.
ranging from simple random extent mutation, to static analysis, and a variety of others. For more information on the techniques currently implemented, please see Section 4.7.

3.6 Summary

In this chapter, we presented an overview of PANDA and described some of its key capabilities as it applies to our research. We also presented a high-level architectural overview of our system and briefly described how the components operate. In the next chapter, we will take this a step further and describe our implementation and the challenges therein.
Chapter 4

Implementation

4.1 Overview

To maximize program coverage, we iteratively employ a number of PANDA runs using the plugins mentioned in Chapter 3. Each stage in this process provides an integral piece of the information necessary to derive new children and forms a feedback loop to drive execution toward greater code coverage. These stages are wholly self-contained and are dependent only on the stage prior. Therefore, we have implemented a queuing system which allows each stage to be processed in parallel with one another to streamline execution time.

In its concrete implementation, each stage communicates with one another via an instance of MongoDB, which is located on the local network. Database records have fields to specify the status of each file as it is processed as well as a prioritization number to indicate precedence. Records also contain relevant information for input requirements and a location for any products, which may include network file system paths. This is to say that each stage is entirely self-contained and the system has been architected to lend itself well to a distributed computation model.

In this model, every stage (with the exception of the merging process) can execute in parallel. It should be noted that the merging process does require mutual exclusion to prevent race conditions and therefore serves as a minor bottleneck. However, our evaluation indicates that the time required for this process is almost negligible in the scope of our overall execution. Overall, this parallel processing model is intended to emphasize practical application and reasonable execution times. For more information on the potential of a distributed system, please see Section 6.1.
Before delving into the specifics, let us first offer a high-level summary of each stage of our implementation. These stages have been annotated on our previous architecture diagram in Figure 4.1.

**Figure 4.1: Annotated Coverage Maximization Architecture**

1. **Ingest Corpus**
   
   This stage simply transfers all test files from the initial input corpus to the Test Files corpus directory. This is a necessary pre-processing stage since it may not be desirable to merge generated children with the initial input files.

2. **Test Files Corpus**
   
   This is the current working corpus of files queued to be processed, including both initial files and children. At this stage, files are simply transferred directly to the “available” queue of the next stage.

3. **Record Test File**
   
   At this stage, a test file is loaded into PANDA and executed as an input to the target program. The entire execution process is recorded and made available for later heavyweight analysis.

4. **Gather Coverage**
   
   This is the heart of the coverage maximization system and where many of PANDA’s capabilities are put to use. These capabilities include OS introspection, taint
tracing, and dynamic value retrieval. At this stage, both branch coverage and
taint information are correlated into one consolidated product. Since this is the
most complicated stage, it is, subsequently, the most time-consuming.


This stage merges the coverage achieved by the current test file into an aggregate
representing the overall total coverage. This is also where we maintain a field
representing the number of attempts targeted at each branch, which is incremented
as necessary.

[4] Pick Conditionals

This stage compares the coverage achieved by the current test file against the cov-
erage frontier to determine which extents should be targeted next. This is where
the scoring algorithm is implemented and where conditional branch targeting pri-
oritization occurs.

[5] Generate Children

Given a set of extents to target and the initial input file, this stage conducts
additional analyses to intelligently choose mutation candidates for child generation.
Generated children are then added to the Test Files corpus for another iteration
and attempt to broaden coverage.

For this research, we have chosen to address targets running on two platforms. The first
of these is a 32-bit x86 image of Debian “Wheezy” with 1GB of RAM. The second is
an ARM image of Debian “Wheezy” with 256MB of RAM (the maximum permitted for
ARM under QEMU). These architectures both correspond to the \texttt{i386} and \texttt{arm} modes of
QEMU, respectively. To maintain consistent program counter (PC) values across runs,
we have disabled address space layout randomization (ASLR) on both systems. This is
not a requirement for our overall technique to work, but is simply used to minimize the
engineering effort required in our implementation.

With that overview, we will now delve into the specific execution at each stage.

4.2 Ingestion and Test Files

When the coverage maximization system first begins, files from the initial corpus are
copied into the Test Files corpus directory. As mentioned in Section 3.5.1, this initial
corpus can contain any set of files with arbitrary contents, but is generally best suited
to a regression test suite.
In addition to copying over the files, this ingestion stage also adds all relevant records to the MongoDB instance to allow the processing cycle to begin.

Using a distinct ingestion process such as this provides two benefits:

1. It prevents any generated children from becoming merged into the initial regression test suite, which may have undesirable effects.

2. It greatly simplifies the process of restarting the system in that it only requires us to clear the Test Files directory and drop the database to reach a fresh starting point. This can be highly convenient for purposes of debugging and optimization.

Overall, the new Test Files directory is simply used as a storehouse from which to fetch the next candidate for testing. It is slowly added to as the system iterates. Manual inspection of this stage can be beneficial for debugging or informational purposes, however the stage itself does very little apart from feeding the Recording process.

### 4.3 Recording

This stage records the whole system as the target program consumes the current input file. To do so, the stage fetches both the file to be analyzed and a path to a directory that will form the basis for a “package”. The contents of this “package” vary with the program being targeted. However, it will always contain either a `run.sh` or `run.bat` script (depending on the guest OS) which will be executed immediately upon being loaded into the guest. The package will also include the target program and all relevant dependencies, as well as utilities to communicate with PANDA via appropriate hypercalls.

To load the package into the guest, the current test file is first copied to the package location on disk. At that time, the entire directory is converted to an ISO image. This image is then virtually inserted into the CD drive of the guest system, where it is identified by a `package_runner.py` script running there. The `package_runner` will then extract the contents of the ISO and execute the `run` script it finds therein.

This process is depicted in Figure 4.2.

A typical `run` script does the following:

First, start a signaler utility to send a custom `hypercall` to PANDA, where it is caught by a custom signaled-recorder plugin. Upon receiving this signal, the signaled-recorder plugin instructs PANDA to begin a recording.
Second, start the panda_taint utility. It accepts the current test file as an argument and labels each byte as tainted. At this stage, the taint plugin is not listening, and will not react to the hypercall the utility performs. However, as every instruction is being recorded, this will be caught during subsequent replay analysis at the coverage stage.

Next, run the target program with the current test file as an input. This is entirely dependent on the target in question and thus at the discretion of the user to manually adjust as necessary. The entire run script process could be automated if it were not for the necessity of this step.

Finally, re-run the signaler utility to signal the end of the recording. The non-determinism log and snapshot are then saved to a configuration-specified location, at which point coverage analysis can begin.

As an optimization, this stage makes use of the QCOW image format’s snapshot capability. With this, we are able to revert the virtual machine to a running state immediately upon starting. This bypasses the boot-up phase and drastically cuts down execution time. This process can also be optimized by placing the QCOW image within a memory-mapped partition - such as a Linux ramdisk. This is currently not implemented in our evaluation, yet is to known to achieve favorable results.

### 4.4 Coverage

At this stage, the recording taken in the previous step can be further analyzed by more heavyweight plugins such as the taint engine. To do so, we fire up PANDA again, this...
time in replay mode.

With the taint plugin engaged, the recording first proceeds in its normal state utilizing the TCG engine. Once it encounters the taint *hypercall* produced by the panda_taint utility, it then switches to LLVM JIT mode and begins to trace the taint on each byte throughout the system.

To retrieve the information we need, it is necessary to correlate the taint data with that provided by other sources, such as the OS introspection and *dynvals* plugins. To do so, we implemented a custom plugin which establishes callbacks and querying functions in the OS introspection, *dynvals*, and taint plugins as necessary.

For instance, we are only interested in branch conditions that take place within our target program. Thus, we need to query the OS introspection plugin to match our expected process name with that of the currently executing process. Interestingly enough, on many Linux kernels, this information can be retrieved from the guest’s virtual memory by calculating a particular offset to the esp register while in kernel mode (see [52] for more on this).

Next, the taint plugin only provides information as it specifically relates to the LLVM taint system. This means that the tainted branch callback only receives the LLVM register number, taint compute number (TCN), and set of taint labels present on that register. Therefore, to fetch additional information (such as the direction a branch was actually taken) we had to employ the *dynvals* plugin and develop a function query it.

From these sources, it is then possible to construct an overall picture of what occurred on each tainted branch condition. This information consists of the register number, TCN, set of extents that produced the taint, taken branch number, and non-taken branch number. This information is then correlated to the starting address of the block in which it occurred and added to a global mapping kept throughout execution.

Once the replay is finished, the aggregated set of branch information objects are marshaled to a file to be merged into the total coverage.

### 4.5 Merging

During the merging stage, the new branch coverage file is integrated into an existing aggregate of total coverage. This total coverage can then be filtered to effectively represent the coverage frontier for the system as a whole. This is implemented via a collection of C++ utilities that provide our data structures and methods to modify, marshall, and merge those structures as needed.
4.5.1 Challenges

Some complications do arise during this stage. For example, merging branch coverage across runs requires that a branch on the same LLVM register in the same function be represented by the same integer value. That is to say, if we conduct two recordings of the target program consuming identical input files, we require all tainted branch conditions to have the exact same register numbers. However, this is not always the case.

While multiple replays of the same recording will always yield identical register numbers (because replay is deterministic), multiple recordings may be subject to minor differences. This is because QEMU implements its own state tracking code at the beginning of each translated block. This information can vary in size across runs dependent on the state of QEMU at that particular moment and thus may introduce additional LLVM registers that were not present in other runs. Since LLVM follows a Static-Single-Assignment (SSA) model, this means that every register following this section will be incremented by a constant amount.

Because the target program’s logic takes place after this QEMU section, each tainted branch we detect will be incremented by the same shift amount on that run. Therefore, we are able to mitigate this issue by implementing a form of fuzzy equivalence. As an example, see Figure 4.3.

![Figure 4.3: Identical LLVM IR Across Two Runs](image)

Other than register numbering, the code in this figure is identical in both runs. Therefore, we can check equivalence by subtracting the offset across runs and verify that it also applies to the taken and non-taken branches. For example, the branch condition from run A has a register number of %78. The branch condition from run B has a register number of %81. To test for equivalence, we first compute %78 - %81 = -3. We can then verify that %88 - %91 = -3 and %81 - %84 = -3 and conclude that the two are equivalent. This allows us to confirm that it was indeed the same branch condition and not another candidate in the same block.
We adopted this approach only because it was relatively simple to implement at the time. As a more robust alternative, we could instead assign an incrementing identifier to each branch condition in the translated block. This works because LLVM instrumentation passes proceed linearly from the beginning to the end of the translated block prior to any execution. To the extent of our knowledge, the QEMU state tracking code never includes additional branches and thus would not interfere with this labeling. The approach could then be implemented in the taint plugin which would pass this identifier rather than register numbers during the on_branch callback. We will more than likely adopt this approach in future work.

4.5.2 Number Attempts

The merging stage is also where we increment any number_attempts fields on existing branches. We increment this field any time the current test file encounters a branch condition that was hit previously by another test file. The merits provided by this technique were described more fully in Section 3.5.3.

4.6 Picking

The picking stage is comprised of a C++ utility that implements the scoring algorithm described in Section 3.5.4. This produces a consolidated report of the branch conditions that should be targeted next and a listing of the input extents that are known to influence them from our taint analysis. This information is then passed to the child generation stage.

4.7 Child Generation

It is up to the child generation stage to accept this information and determine how to intelligently mutate the extents affecting those branches. A variety of strategies can be applied at this stage, all of which vary in complexity and have associated pros and cons. More complex strategies have the potential to be more compute-intensive and slower. However, they are also more likely to yield additional coverage and therefore receive a higher prioritization in subsequent iterations of the system.
4.7.1 Random

Random mutation is the most simple to apply and also the least likely to yield additional coverage. Unlike traditional full-file fuzzing though, we only apply this strategy to extents that are known to taint the condition of the candidate branch. This method is ultimately far more precise than full-file fuzzing and serves as the fall-back option if more sophisticated strategies fail to make progress.

4.7.2 Educated Guesses

This approach is similar to the “Known Integers” technique applied in American Fuzzy Lop (AFL), COMET, and other systems [10, 46]. It consists of specific hard-coded values that have an increased probability to trigger edge-conditions in the targeted code. These values include integer constants such as 1, 0, and -1 (0xff).

This is marginally superior to random mutation and therefore receives slightly higher prioritization.

4.7.3 Static Values

This technique was also used in COMET, where they extracted what they referred to as “static-hints” [10]. In their implementation, these “hints” were any hard-coded values being compared to at the branch condition.

Since we are not provided with source code, we instead directly parse the text of the LLVM IR that a candidate branch took place in. From this text, we extract any values that could be construed as hard-coded comparisons. This would include extracting “10” for the “if (x==10)” dilemma previously described in Section 2.1.1.

This simple text-parsing strategy is not currently correlated to the precise extent it applies to. However, this could trivially be implemented in the form of a basic backward-slicing algorithm. Instead, any extracted values are simply attempted at each potential tainted extent location.

Ultimately, this technique begins to demonstrate some of the power provided by further analysis of the LLVM IR. This strategy alone can yield vastly increased coverage in certain situations such as comparison with integer or character constants, or comparison with magic file headers. For more on this, please see our evaluation in Chapter 5. At the moment, this technique is the most complex strategy in place and therefore has the greatest priority for subsequent iterations.
While these are currently the only mutational strategies featured in our system, there is significant potential for further improvement. Additional strategies which seem promising include dynamic value extraction through further LLVM instrumentation as well as the potential to integrate with symbolic execution engines such as that used by KLEE [8]. For more information on this, please see Section 6.1.
Chapter 5

Evaluation

5.1 Methodology

To evaluate our system, we conducted two separate experiments and measured any increases in program coverage.

As an adversarial model, we compared our performance with the relatively simple technique of random file generation. Specifically, we used the `dd` utility to dump uniformly random test files from `/dev/urandom`. The length of these files differs from test to test but generally conforms in size with that of a “well-formed” input for the target in question. To test random files, we iteratively generated a new file and then tested the target with that file immediately. That is to say that the random files were not generated a priori, and therefore the time required to produce them should be taken into account in our results. This was done to be consistent with how a true adversary employing random file fuzzing would operate.

For both of our experiments, we compiled the target using GCC on the default optimization level. We executed the random tests on the host machine itself outside of the scope of QEMU with full processing power. We then measured coverage for distinct durations in each experiment and compared the outcomes.

At this point, it is worth discussing our method of gathering coverage metrics. Namely, we decided to record metrics outside the scope of our framework itself. Generally, it is helpful that our strategy targets the true coverage of the binary by analyzing branches at the assembly level. This can overall have a net positive effect when comparing the efficacy of child generation strategies and is more accurate than source line coverage which fails to account for any compiler-introduced inconsistencies. However, there are a few issues with this approach that make comparison difficult.
First, it is challenging, if not impossible, to achieve a reasonable upper-bound on the total number of blocks that could be covered by the target program. Unlike source code coverage, where the number of lines is known, there is no way to determine at an assembly level how many blocks a program has the potential to execute. This occurs for a number of reasons, including the presence of indirect branch conditions, which inhibit the ability to count blocks via static analysis. Thus, the only metric we could achieve is a raw count of the number of blocks that were covered during a run. This comparison is ultimately not very informative and does not help identify specific cases where our approach may fail.

Next, as the results below demonstrate, the recording process itself can be relatively time consuming. While we could establish metrics for our own test files during the course of execution, retrieving the same results for every randomly generated file would take a significant amount of time. For example, if each recording were to take 20 seconds then the approximately 1 million random test files generated in our first experiment would take around 231 days to process.

Therefore, for the purposes of our results in this research, we measured coverage at the source line level via the `gcov` measurement tool. This provides an upper-bound on coverage and gives a reasonable level of accuracy to compare performance.

Both experiments were performed on a single workstation with eight 2.3GHz AMD Opteron processor cores, 32GB of RAM, and running on Linux kernel 3.2.0-4.

### 5.2 Static Value Extraction Testing

For our first experiment, we propose the following contrived example:

```c
// Read input into buf
if (buf[0] != 'P') exit(0);
if (buf[10] != 'a') exit(0);
if (buf[20] != 'N') exit(0);
if (buf[30] != 'd') exit(0);
if (buf[40] != 'A') exit(0);
// Done
```

In this example, various bytes of the input file are compared with constant values before being permitted to continue. This simulates a real-world scenario in which an input is forced to pass initial format conformance checks prior to entering the core logic of a parsing application. Comparisons such as this are often used to check magic-number headers or other format-dependent bytes to ensure that an input file is valid before
wasting precious processing time to fully parse it. This also demonstrates the “if (x ==
10)” case mentioned in Section 2.1.1.

The intent behind this example is to demonstrate the static value extraction child-
generation technique currently utilized by our framework. The above snippet was at-
tached to some bootstrapping code which simply read the input file from argv into buf. Therefore, the application in its entirety came to 33 lines of code as measured by gcov.

Our initial corpus was composed of a single input file with 64 uniformly random bytes, none of which were initially aligned with the values being compared in the target. Files created by the random generation strategy were also 64 bytes in length. We compiled this target for both 32-bit x86 and ARM using GCC.

5.2.1 Results

The results for this experiment are presented in Figure 5.1:

These graphs show that our our approach was able to achieve 100% coverage in a rel-
atively small number of attempts. It was able to extract the relevant constants from the LLVM IR at each step and apply them in the next stage of child generation. In this way, we achieved 100% coverage through a depth of five nested branches in only 29 input files for both x86 and ARM. Additionally, because of the prioritization of static-
value-derived children, these results are almost always identical across experiments. The only exception would occur if a randomly mutated file happened to yield the appropriate value, which is statistically unlikely. Therefore, error bars have been intentionally omitted from these results.
As an aside, it should be noted that many of the 29 files which were generated were the result of parallel analyses still processing alternate paths to the goal. This number would change if we were to linearly process each test file.

Once the final half-covered conditional became fully covered, the system intelligently halted and displayed the following x86 total coverage:

```
134514275 -
    BranchInfo(r_num: %92, c_num: 2, t_branch: 98, nt_branch: 93,
        num_attempts: 0, i_extents: { 40, })
    BranchInfo(r_num: %92, c_num: 2, t_branch: 93, nt_branch: 98
        num_attempts: 0, i_extents: { 40, })
134514238 -
    BranchInfo(r_num: %92, c_num: 2, t_branch: 98, nt_branch: 93,
        num_attempts: 1, i_extents: { 30, })
    BranchInfo(r_num: %91, c_num: 2, t_branch: 92, nt_branch: 97
        num_attempts: 5, i_extents: { 30, })
134514201 -
    BranchInfo(r_num: %91, c_num: 2, t_branch: 97, nt_branch: 92
        num_attempts: 7, i_extents: { 20, })
    BranchInfo(r_num: %92, c_num: 2, t_branch: 93, nt_branch: 98
        num_attempts: 5, i_extents: { 20, })
134514164 -
    BranchInfo(r_num: %92, c_num: 2, t_branch: 98, nt_branch: 93
        num_attempts: 13, i_extents: { 10, })
    BranchInfo(r_num: %92, c_num: 2, t_branch: 93, nt_branch: 98
        num_attempts: 5, i_extents: { 10, })
134514124 -
    BranchInfo(r_num: %124, c_num: 7, t_branch: 130, nt_branch: 125
        num_attempts: 19, i_extents: { 0, })
    BranchInfo(r_num: %127, c_num: 7, t_branch: 128, nt_branch: 133
        num_attempts: 8, i_extents: { 0, })
```

Key:

- **r_num**: LLVM Register Number
- **c_num**: Taint Compute Number
- **t_branch**: Taken Branch
- **nt_branch**: Non-taken Branch
- **num_attempts**: Number attempts at branch condition
- **i_extents**: Input extents affecting the branch

This coverage identifies the five distinct branch conditions that were encountered during the course of execution. It is interesting to note that each branch condition was also placed in its own translation block as designated by the program counter (PC) value corresponding to the first instruction in that block. These values are displayed in decimal, yet may be more recognizable in hexadecimal as typical user-space addresses in the 0x0804 range: 134514123 = 0x080485cc.

This listing also demonstrates the register shifting phenomenon that takes place between runs as described in Chapter 4. In particular, the branch in translated block 134514124
shows a condition dependent on both register %124 and %127. However, further inspection reveals that both actually reflect the same branch. This can be verified by noting that there is a numeric shift of 3 that can also be seen in the \texttt{t\_branch} and \texttt{nt\_branch} values.

The winning candidate that bypassed all checks was named \texttt{randfile\_1.0.0.0.0} following a naming convention in which each child is assigned an identifier that is appended to the name of its parent. Thus, the 5 digits indicate that five generations were produced corresponding to the 1st, 0th, 0th, 0th, and 0th children respectively. The low numbers in these identifiers are a result of prioritizing the static value extraction strategy in child generation.

The first generation extracted a static value from the LLVM IR that was unrelated to the branch condition and therefore made no progress. Its sibling (with id=1) did manage to succeed however, and went on to yield the final solution. Thus, if we had executed our strategy linearly, it would have only taken 5 test files to achieve 100\% coverage.

The average times taken at each stage for x86 are listed in Table 5.1.

\begin{table}[!h]
\centering
\begin{tabular}{|l|c|}
\hline
 & \textbf{Execution Time (s)} \\
\hline
Recording & 21.4 \\
Coverage & 250.8 \\
Picking & 0.03 \\
Generation & 0.03 \\
\hline
\textbf{Total:} & 29m21s \\
\hline
\end{tabular}
\caption{Average x86 execution time for static value tests}
\end{table}

It took slightly longer to process this test under ARM, which achieved an identical outcome in a total time of 48 minutes, 15 seconds. There are a number of factors that could contribute to this and further work is needed to identify possible optimizations.

After this test, we then allowed a random generation strategy to execute for the same total amount of time using all available processing power. During this time, more than a million tests were generated. However, the third branch was never overcome. This ultimately demonstrates the limitations inherent in black-box techniques and motivates the cause for introspection-guided approaches to broaden coverage.

It is worth noting that our success in this test is primarily owed to two factors. First, the taint system allows us to know exactly which bytes of the input file should be replaced by any static values extracted. Second, this extraction is only possible because individual characters are encoded as constant integer values within the assembly code itself. For example, the 'P' character corresponds to the ASCII digit 80 within the LLVM IR. This integer encoding is what allowed us to extract these values from the text of the IR for
child-generation purposes. This is to say that our current stage of child generation would not catch these values if a more robust string comparison algorithm such as `strcmp` were used.

The case of `strcmp` can be complex because of the variety of ways in which it can be implemented. For example, the SSE4.2 instruction set included four new XMM instructions specifically intended to speed up text processing code. For a detailed example of how these instructions are used, please see [54]. Generally, this means that it can be difficult to derive a reasonable value to extract, particularly within highly optimized code. For more details on potential solutions to this hurdle, please see Section 6.1.

Ultimately, our results in this test motivate the cause for grey-box fuzzing strategies and demonstrate the potential for more heavyweight analyses to reach deeply within the nested logic of an application.

### 5.3 Sendmail Excerpt Testing

In our next test, we applied our strategy to a historical excerpt from sendmail. This excerpt was taken from an earlier evaluation of static analysis buffer overflow detectors and was also featured in COMET's research evaluation [10, 55, 56]. The example itself was labeled `s1` in the COMET research and demonstrates a known buffer overflow. As the intent of our research is to eventually produce a robust fuzzing platform, this seemed an ideal candidate to test our fledgling child generation capabilities.

We did not modify the target in any way from the original models in [55, 56]. The application simply accepts an input via `stdin` and supplies it for processing by a faulty `crackaddr` function call. In total, `gcov` measures the application at 181 lines. To adapt this target for our test harness, we simply changed the `run.sh` script to redirect the current test file as standard input to the target application. No other changes were necessary.

Our initial corpus in this test was 20 input files varying in length from 4 to 128 uniformly random bytes. The tests were conducted assuming zero knowledge of the input, including the expected length. Random files similarly varied in size from 4 to 128 bytes in length.

We compiled and ran this test for x86 with and without static value extraction, and for ARM with static value extraction. We allowed both strategies to process for two hours, as dictated by the Linux `timeout` utility. We then gathered coverage for all files in the corpus, including those that had been generated but not yet fully processed by our system. We then compared the relative coverage achieved by each strategy.
5.3.1 Results

The results from this experiment are presented in Figure 5.2.

**Figure 5.2:** S1 coverage test results

There are several items of note in these results.

First, these tests show that the static value extraction strategy is highly beneficial in real-world application as well as in our last toy example. In this case, it improved overall coverage from the 87.85% without static value extraction to 90.61% with the extraction. In this, it found cases such as “if (c == '('). Without this capability, these branches would rely on random chance to be covered. The prioritization of these static values is also visible in the graph as these tests immediately increase coverage with a rapid jump near the beginning of execution. Testing without static value extraction eventually uncovered some of these paths through random chance, but required significantly more effort.

Next, this graph also shows that the ARM system functions just as well as the i386 system. In this case, it actually fared slightly better and achieved 91.16% coverage overall. This may be indicative of the architecture composites concept that we introduce in Section 6.1.5.

Finally, these tests also show that, while our strategy was able to make some headway, it did not outperform random testing over the same amount of time. In two hours, our strategy progressed from the 66.30% coverage achieved by the initial corpus to 90.61% in total. In the same amount of time, the random mutation strategy progressed to a total of 94.98%.

There are a number of reasons to explain our lack of progress. However, the most basic is that our current child generation strategies are simply not powerful enough to extract
profitable information from the translated LLVM IR. For further ideas to improve upon this, see Section 6.1.

To date, we have not run this test for longer than two hours to see if our strategy eventually outperforms random testing. However, this is intended as a step in the near future.

Our approach recorded a large number of tainted branches within each of 393 different translated blocks. Taint propagation was much more complex in this example than in the previous contrived test and can be seen in the increased amount of time taken during the covering stage. The average times taken at each stage are listed in Table 5.2.

<table>
<thead>
<tr>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recording</td>
</tr>
<tr>
<td>Coverage</td>
</tr>
<tr>
<td>Picking</td>
</tr>
<tr>
<td>Generation</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
</tr>
</tbody>
</table>

Also notable from this table is the increased execution time taken by both the picking and generation stages. While this is almost negligible in comparison to the time taken by coverage, it is still worth noting the correlation between the number of branches being reasoned over and the total execution time.

5.4 Conclusions

Preliminary tests with more robust real-world samples are promising in that they are readily adaptable to the current test harness and require no modification or reliance on source code.

However, initial results show that our current implementation does not make significantly more progress than that produced by random testing in a two-hour time frame. The techniques that we apply to child generation are simply not powerful enough to extract the necessary values needed for future children to explore greater volumes of code.

This is not at all to say that our results are unsatisfactory. Instead, our tests in static value extraction are highly auspicious and prove that the methods we employ have significant opportunity to increase program coverage. We are only marginally tapping into the wellspring of potential offered by this approach and further improvements appear promising with oncoming future work.
Chapter 6

Conclusions and Future Work

In this chapter, we review what we have accomplished in this research and identify key areas in which to improve upon our results.

6.1 Future Work

As the evaluation stage revealed, we are currently only glimpsing the potential that could be afforded by our strategy. We eventually intend to produce a robust fuzzing framework that is both capable and practical. To this end, future work could be directed at one of several objectives.

6.1.1 Test Set Minimization

When developing regression test suites, it is often desirable have a minimum set of tests that will fully exercise the entire functionality of the target program. This lessens the overall amount of time (viz. money) required to run the tests and also simplifies the testing process overall. With our current approach, it should be feasible to identify the test cases that exercise a specific branch and, subsequently, derive the entire path that the file took. In this way, it should be relatively simple to extend our approach to identify the set of files that exercise the entire functionality of the target with minimal overlap. In its most basic implementation, this could simply be a greedy algorithm that constructs a corpus as our technique runs. More complex approaches should be possible as well with little-to-no modification to our current architecture.
6.1.2 Practicality

First, while the COMET project’s evaluations took place on a scale of seconds, our work takes place on the scale of minutes. This is obviously non-ideal. However, our approach also provides significantly broader capabilities including the non-requirement of source code and whole-system taint analysis. Thus, we could make the argument that “with great power comes great execution time”.

However, there are steps that can be taken to achieve better coverage in less time and would allow this work to extend to much larger applications than those currently tested. Trivially, as mentioned in Chapter 4, this framework lends itself well to a distributed computation model. While our current implementation strictly executes using multiple processes on a single machine, this could easily be extended to a cluster with very little adaptation of our current source code.

Next, one of the greatest issues facing our framework are the long queues of input files awaiting processing. Distributed computation may alleviate this, however it may also just as likely exacerbate the problem. Instead, more progress could be made by integrating a corpus culling algorithm as used by many modern fuzzers including Peach and American Fuzzy Lop [46, 57].

If we were to adapt the AFL model, then we would periodically re-evaluate the input corpus to cull out any input files that have been superseded by newer candidates. To do so, we would simply track the branch conditions that each file is intended to address and identify if it is a strict subset of the branches now being targeted by later candidates. Thus, we could greatly lessen the number of files that would otherwise only serve as a waste of computation time. For a more detailed explanation of how this algorithm could function, see the technical notes in Section 4 in [58].

Finally, in the most basic way, there is significant room for optimization within our source code itself. While this work gives a general idea of the magnitudes that should be expected for execution time, it does not at all suggest a lower-bound.

6.1.3 Fuzzing Techniques

While this work is intended provide a foundation for a fuzzing platform, its security capabilities have thus far been largely neglected. Thus, candidates for future work include numerous heuristic techniques that could be applied to operate more distinctly as a fuzzer.
For example, it should be trivial to implement functionality that would allow greater targeting of specific libraries and functions such as that utilized in BuzzFuzz [9]. This could be broadly scoped toward user-definable points of interest or simple heuristics to identify commonly vulnerable functions. PANDA has the capability to identify memory regions within a target and thus the construction of additional utilities to pinpoint individual libraries or functions would be relatively simple.

It could also be trivial to integrate various forms of bug triaging and exploitability testing. While these are generally open fields for research, basic capabilities have already integrated during the recording stage described in Section 4.3. This current functionality ultimately amounts to a hypercall if the target program yields a non-zero exit code, however there should be no limitations in employing other methods.

As an alternate approach to branch coverage tracing, it may also be worth employing the block-transition coverage technique utilized in AFL [46]. This could form a great supplement to our existing tainted branch tracking and help prioritize promising test files that traverse the code base in unique ways.

Finally, in addition to considering some of AFL’s strategies, it may be worthwhile to integrate with AFL (or some other lightweight product) as a whole. Their technique executes very quickly and covers a significant portion of the target’s code before being forced to resort to random mutation. Alternatively, our technique operates relatively slowly but can identify the key input combinations necessary to make it beyond previously insurmountable branch conditions. These two approaches form an ideal complement for practical vulnerability discovery. More information on this process is described in [46].

### 6.1.4 Child Generation

As our evaluation revealed, simple static value extraction is not powerful enough to make headway in real-world applications. Instead, we should employ more varied techniques and utilize some of the power provided by the translated LLVM IR.

#### 6.1.4.1 Multiple Extent Correlation

First, there are basic improvements we can make to the types of mutations we generate. Our current implementation only considers tainted input extents on a byte-by-byte basis. That is to say, we do not currently correlate clusters of bytes or make any attempts to derive broader overarching concepts from the contents of an input file. There has been significant research in this, including both automatic grammar generation and reverse engineering of protocol formats as described in Chapter 2 [34, 35].
However, a more reasonable first step in this direction would be in simply mutating clusters that appear close together. For example, if a branch condition relies on extents 0..3 then all extents should be simultaneously mutated, possibly via some reasonable heuristic.

### 6.1.4.2 Handling Strcmp

Next, there is the dilemma of how to handle string comparisons such as those performed by `strcmp`. In its most basic form, we could simply identify a library call to the function in a relevant version of `libc`. However, this neglects cases where the function call may be in-lined, statically compiled, or optimized. To combat this, we could then turn to some heuristic form of multiple extent correlation. In other words, if a branch condition is tainted by extents 4..10 then we could make some assumptions that it is subject to a variant of `strcmp` or possibly `memcmp` because it is based on a wide range of values. However, this approach is very naive and still does not supply us with reasonable values to apply in child generation. For example, `strcmp` and `memcmp` generally operate on pointers and have no applicable static values to extract. Instead, the solution may reside in dynamic value extraction.

### 6.1.4.3 Dynamic Value Extraction

The COMET system instrumented the source code of the original program and inserted functionality to log what the authors referred to as “dynamic-hints” \[10\]. These “hints” contained the value at runtime of specific variables as they were used in a branch condition. Through our access to the LLVM IR, we could implement something very similar.

This type of functionality would allow us to solve entire categories of problems that are otherwise near-impossible for traditional black-box techniques. For example, consider the triangle problem \[10\].

In the triangle problem, three integers are provided to the target program. This target then determines whether or not they would make a valid triangle if they were considered as the lengths of the sides. If they are valid, then the program yields the type of triangle that they would form (ex. isosceles, equilateral, or scalene). This problem is generally difficult for black-box fuzzers because they are unable to comprehend the relationships between components. However, this problem was trivial for a system such as COMET because of its ability to dynamically extract the values that are being compared and use them in child generation.
We could implement a system similar to this by employing another run of PANDA during the generation stage. During this run, we would utilize a plugin specifically designed to dump the contents of relevant LLVM registers within a specified translated block. This approach would theoretically consume much less time than our current coverage stage because it does not require continued LLVM translation like that required by the taint system.

Ultimately, this could be applied in a variety of ways. For example, it could be used to address the complications with `strcmp` by extracting the pointers being compared. PANDA can then read this pointer in the guest’s memory and yield a useful value for future tests.

Dynamic value extraction is a highly promising technique and should be considered as a next step in future work.

### 6.1.4.4 Symbolic Execution

Finally, the most powerful technique we could employ would involve integrating a symbolic execution engine such as that provided by KLEE. Because KLEE operates directly on LLVM IR for its analyses, it pairs well with our current model and is a very promising candidate for future research.

Generally, approaches utilizing symbolic execution tend to be computationally infeasible in most practical use cases. However, this has been partially addressed by employing techniques such as concolic execution. The recording process that we employ in our research roughly parallels the concolic process. Additionally, with KLEE, it should be possible to symbolically execute only the input extents that are of interest in the current test. The tractability of this has been demonstrated in previous research, including in Sean Heelan’s thesis. In his research, he discovered that the symbolic execution of limited set of specific bytes, as derived from taint tracking, proved to be highly feasible [36].

It should be possible to integrate an analysis such as this with another PANDA run during the generation stage. In theory, an approach such as this could be extremely beneficial. However, further work will be required to assess the practical integration of these tools.
6.1.5 Architecture/Optimization Composites

A final novel contribution of this work (as far as we are aware) is in the description of a method to increase program coverage by consolidating analyses of a target application which is compiled for multiple architectures or at different optimization levels.

Even after sophisticated analyses such as those described above are applied, there will undoubtedly be cases in which the JIT-translated LLVM will simply not supply enough information to intelligently choose a child candidate. This situation will likely arise from some nuance in the control flow graph of the target application which limits the amount or quality of information present at a certain block.

However, we believe that this can be overcome by executing the same target program on an alternate architecture or at a different optimization level. The alternate compilation target will inevitably have a different internal structure than the original and thus will translate differently into LLVM IR. Therefore, given the same input file, it will approach what is effectively the same branch condition from a perspective that may be more profitable.

For example, if we are unable to make headway on the mupdf program on the x86 architecture, it may help to execute an ARM-compiled version, or, alternatively, to execute versions of mupdf at both -O0 and -O3 optimization levels. Since both targets utilize the same source code, they should have the same capabilities and therefore the same branch candidates.

The alternate architecture approach works particularly well for PANDA because of its standardized analysis medium via LLVM IR. Our current system supports both ARM and x86 and could easily employ this approach by sharing the Test Files queue and independently marking each file as complete for each architecture. The feedback loop would then be driven from both targets and could potentially find more favorable child candidates in one case or the other. More work is needed to verify this, however the concept appears promising.

6.2 Conclusions

In this work, we have expounded upon a novel method that can be used to maximize the test case coverage of a target program with almost no a priori knowledge. We have demonstrated some of the capabilities that our approach provides and have also identified key areas for improvement for future work. Ultimately, the contributions of this work are:
• A platform to conduct grey-box dynamic analysis of a program and identify specific input bytes which influence branch conditions within that program.

• A method to measure 100% “true” instruction coverage, rather than 100% source coverage to unmask compiler introduced discrepancies.

• Description of a metric to determine when test file mutation may reasonably halt.

• Description of a scoring method to prioritize input files most likely to yield increased coverage.

• Description of a method to compose analyses of a target program compiled for multiple architectures or optimization levels to derive otherwise inextricable information.

Hopefully this work will serve as a small yet integral step in greater things to come.
## Appendix A

### Coverage Status Console

---

#### TESTFILES

<table>
<thead>
<tr>
<th>timestamp</th>
<th>_id</th>
<th>name</th>
<th>prio</th>
<th>status</th>
<th>loc_on_disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>02Mar15 08:10:41</td>
<td>54f46151724445ccac1584e0</td>
<td>mmmm</td>
<td>5</td>
<td>finished</td>
<td>./test_files/mmmm</td>
</tr>
<tr>
<td>02Mar15 08:10:41</td>
<td>54f46151724445ccac1584e1</td>
<td>aaaa</td>
<td>5</td>
<td>finished</td>
<td>./test_files/aaaa</td>
</tr>
<tr>
<td>02Mar15 08:10:41</td>
<td>54f46151724445ccac1584e2</td>
<td>xxxx</td>
<td>5</td>
<td>finished</td>
<td>./test_files/xxxx</td>
</tr>
</tbody>
</table>

Current Time: 02Mar15 08:11:23

---

#### RECORDING

<table>
<thead>
<tr>
<th>timestamp</th>
<th>_id</th>
<th>name</th>
<th>init_id</th>
<th>prio</th>
<th>status</th>
<th>loc_on_disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>02Mar15 08:10:43</td>
<td>54f46153724445ccac1584e5</td>
<td>xxxx</td>
<td>54f46151724445ccac1584e2</td>
<td>5</td>
<td>available</td>
<td>./test_files/xxxx</td>
</tr>
</tbody>
</table>

Available -- 1

In-use -- 1

---

50
Appendix A. Coverage Status Console

Finished -- 1
02Mar15 08:10:41 54f46151724445ccac1584e3 mmmm 54f46151724445ccac1584e0 5
finished ./test_files/mmmm

===============================================
2. COVERING
===============================================
KEY: timestamp _id name init_id prio status rec_path

Available -- 0
In-use -- 1
02Mar15 08:11:05 54f46151724445ccac1584e6 mmmm 54f46151724445ccac1584e0 5
in-use ./recordings/i386_linux_sv_multi_test_mmmm
Finished -- 0

===============================================
3. MERGING
===============================================
KEY: timestamp _id name init_id prio status br_cov_path

Available -- 0
In-use -- 0
Finished -- 0

===============================================
4. PICKING
===============================================
KEY: timestamp _id name init_id prio status tot_cov_path new_cov_path

Available -- 0
In-use -- 0
Finished -- 0

===============================================
5. GENERATING
===============================================
KEY: timestamp _id name init_id prio status templ_path

Available -- 0
In-use -- 0
Finished -- 0
Appendix B

Available PANDA Plugin Callbacks

// Before translating each basic block
PANDA_CB_BEFORE_BLOCK_TRANSLATE

// After translating each basic block
PANDA_CB_AFTER_BLOCK_TRANSLATE

// Before executing each basic block
// (with option to invalidate, may trigger retranslation)
PANDA_CB_BEFORE_BLOCK_EXEC_INVALIDATE_OPT

// Before executing each basic block
PANDA_CB_BEFORE_BLOCK_EXEC

// After executing each basic block
PANDA_CB_AFTER_BLOCK_EXEC

// Before an instruction is translated
PANDA_CB_INSN_TRANSLATE

// Before an instruction is executed
PANDA_CB_INSN_EXEC

// After each memory read (virtual addr.)
PANDA_CB_VIRT_MEM_READ

// Before each memory write (virtual addr.)
PANDA_CB_VIRT_MEM_WRITE

// After each memory read (physical addr.)
PANDA_CB_PHYS_MEM_READ

// Before each memory write (physical addr.)
PANDA_CB_PHYS_MEM_WRITE
/ Hypercall from the guest (e.g. CPUID)
PANDA_CB_GUEST_HYPERCALL

// Monitor callback
PANDA_CB_MONITOR

// On LLVM JIT initialization
PANDA_CB_LLVM_INIT

// In cpu_restore_state() (fault/exception)
PANDA_CB_CPU_RESTORE_STATE

// before system call
PANDA_CB_USER_BEFORE_SYSCALL

// after system call (with return value)
PANDA_CB_USER_AFTER_SYSCALL
Bibliography


[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Security and Privacy (SP), 2010 IEEE Symposium, pages 317–331. IEEE, 2010.


