ENHANCING DIRECTED SEARCH IN BLACK-BOX, GREY-BOX AND WHITE-BOX FUZZ TESTING

VAN-THUAN PHAM

(M.Eng., HUST)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2017

Supervisor:

Professor Abhik Roychoudhury

Examiners:

Associate Professor Liang Zhenkai

Dr Prateek Saxena

Associate Professor Cristian Cadar, Imperial College London

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Pham Van Thuan

15 June 2017

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Dr. Abhik Roychoudhury, for his wonderful support and guidance during my PhD endeavors. I will recall the weekly meetings when I just joined his group and got his invaluable advice on formulating research questions and communicating ideas in a concise yet effective way. His insights and continuous feedback have shaped the core ideas for all of my work. More importantly, his passion for doing research influences my working attitude during my PhD studies as well as in my future career. Without his support and guidance, this thesis would not have been possible.

I wish to thank all of my family members especially my parents, my wife, my little daughter and my brothers for their unconditional love and support. I thank my wife for her understanding as well as her encouragement whenever I have difficulties at work. Without her endless love and support, I would not have been able to complete my PhD journey.

I am thankful to Dr. Liang Zhenkai and Dr. Prateek Saxena for agreeing to serve in my thesis committee and giving constructive comments on my thesis proposal. I would also like to thank Dr. Cristian Cadar for taking his precious time to be my external thesis examiner.

My study in NUS could not have been so wonderful without my colleagues, collaborators and friends. I would like to thank Sudipta, Lee Kee, Clement, Konstantin, Wei Boon, Subhajit, Saakar, Jooyong, Chia Yuan and especially Marcel for their productive discussions and great collaborations. I would also like to thank my friends and my labmates, Manh Dung, Quang Loc, Duc Hiep, Quang Trung, Ton Chanh, Minh Thai, Duy Khanh, Truong Khanh, Abhijeet, Sergey, Shin Hwei for inspiring discussions about research topics and the life.

The work presented in this thesis was partially supported by a grant from DSO National Laboratories, Singapore, and the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCRNCR001-21) and administered by the National Cybersecurity R&D Directorate. They are all gratefully acknowledged.

PAPER APPEARED

1. Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing failing tests via symbolic analysis. *20th International Conference on Fundamental Approaches to Software Engineering (FASE) 2017. pp.43–59.*

2. Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2016. pp.543–553.*

3. Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: reproducing crashes in real-world application binaries. *In Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15). pp.891–901.*

Table of Contents

Enhancing Directed Search in Black-box, Grey-box and White-box Fuzz Testing

Abstract

Security bugs can exist in every single software system, and software testing aims to find these bugs ahead of attackers, who are hunting for zero-day bugs and/or managing to exploit them for profit (e.g., by stealing users' credentials like credit card information) or to cause serious problems (e.g., by attacking critical systems like nuclear power plant). Fuzz testing (or fuzzing) techniques, which include (model-based) black-box, coverage-based grey-box and white-box approaches, have become prominent for software testing. However, given an *inadequate test suite* they are not skillful at directing the exploration to reach *given target locations* and expose bugs in *large program binaries* that take *highly-structured* inputs. We observe that these limitations can be circumvented by improving the directedness of fuzzing approaches.

In this thesis, we first design a directed search algorithm in Hercules, a symbolic execution based white-box fuzzing engine working directly on large multi-module (stripped) program binaries. The directedness of Hercules is attributed to its ability to steer the exploration towards target locations using the module dependency graph and control flow graph lifted directly from application binaries. Moreover, by exploiting the results produced by SMT constraint solver (e.g., minimal unsatisfiable core), Hercules systematically navigates the search between non-crashing paths and crashing ones. White-box fuzzing tools like Hercules excel at reasoning about values of data fields but it could easily get "stuck" at synthesizing the whole (optional) data block (a.k.a data chunk) which may not exist in an inadequate test suite of highly-structured inputs like

PNG, WAV and PDF files. To tackle this problem, we develop MoBWF — a novel combination of model-based black-box fuzzing (as embodied by Peach fuzzer) and white-box fuzzing (as embodied by Hercules). In this combination setup, Hercules can inform Peach about where it gets stuck. Peach takes this information and leverages the input model to generate and transfer the missing data block to the current input of Hercules, helping Hercules get unstuck and continue its directed exploration. Apart from expensive symbolic analysis based approaches, the directedness can also be achieved by augmenting coverage-based grey-box fuzzing (CGF), a more lightweight technique. We build AFLGo, a directed CGF, by integrating Simulated Annealing global search algorithm into the fuzzing process so that the testing is steered towards target locations with a higher probability than other locations.

The experimental evaluations on two applications of directed fuzzing – crash reproduction and patch testing for vulnerability detection – show that Hercules, MoBWF and AFLGo effectively guide the search and successfully reproduce crashes in large real-world (binary) programs (e.g., Adobe Reader, Windows Media Player, OpenSSL) taking highly-structured file formats (e.g., PNG, WAV, PDF). Notably, AFLGo can expose the famous HeartBleed vulnerability almost four (4) times faster than the state-of-the-art AFL fuzzer. In addition, AFLGo has discovered 14 zero-day vulnerabilities in the widely-used Binutils toolset. All the vulnerabilities have been confirmed and fixed by Binutils' maintainers, and we have obtained five (5) CVEs assigned to the most critical ones.

List of Tables

List of Figures

Chapter 1

Introduction

Computing devices are omnipresent in our everyday life. As of October 2014, there were about 2 billion of personal computers (including PCs and Macs) and more than 7 billion of handheld devices in use worldwide [12]. This figure will be much bigger if we also count the embedded devices in this era of internet of things (IoT). Millions of software applications are running to control the devices and provide utilities to end users. In this situation, any bug/defect in a (critical) software can lead to costly remedial actions. A software application can have different types of bugs. A bug can be simple and harmless like the one which only affects user experiences (e.g., incorrect font or improper display screen size). More seriously, a logical bug can cause programs to output unexpected results. From a security perspective, most severe bugs stop a software system from running properly (e.g., denial of service) or pave the ways for attackers to install and execute malicious code which can steal users' confidential data (e.g., credit card information). For example, the Heartbleed bug [103] in the popular OpenSSL cryptographic software library caused secret key leak, and allowed anyone on the Internet to steal data directly from the compromised services. Even worse, malicious code can control the whole

software and physical system. In December 2015 a malware, which has recently been dubbed "Crash Override" [1], showed its ability to control the Ukraine power grid and disrupt electricity supply to the end consumers. The cyberattack left 230 thousands people without electricity in several hours[13].

To achieve high level of correctness and security in software applications, software manufacturers should continuously conduct three steps – testing, debugging and fixing through the whole life of their software products, during development process and even after the software is released. In these three steps, testing plays a key role to produce test cases proving the presence of software defects. Once such a test case is generated, developers can debug the buggy program, find its root cause and fix it.

Software testing can be done manually, semi-automatically or automatically. Even though manual testing is still a common practice in software development and maintenance process, it is a time and human resource consuming task. Nowadays software systems are getting larger and more complicated, and it makes manual testing much more challenging. Hence, (semi) automatic testing has been in high demand especially when computing resources are getting more accessible and cheaper than ever as a result of significant advances in Cloud computing technology.

Several automatic testing techniques have been proposed, and we can categorize them into static and dynamic approaches. Static techniques include static analysis [20, 6, 64] and software model checking [55, 38, 18, 37]. While static analysis tools perform without executing the program under test (PUT), model checkers work on a model of the PUT. Although static approaches have shown their effectiveness in finding

[1] https://dragos.com/blog/crashoverride/

program bugs, they can only indicate potential buggy locations and produce program traces to these locations – they cannot produce concrete test cases to trigger the bugs. Consequently, developers/testers still need to spend substantial time and efforts to check whether the reported bugs (i.e., alarms from static analysis tools) are real bugs. Indeed, static approaches suffer from high false positive rate, that is, they usually raise lots of false alarms because of lacking run-time information or the imprecision of the program model. In contrast, dynamic approaches (e.g., Fuzz testing [9, 4, 5, 35, 51, 8, 101]) execute the PUT and generate concrete test cases as witnesses for program bugs. There is minimal to no effort needed by developers to validate the bugs reported by dynamic tools.

In the scope of this thesis, we focus on dynamic approaches particularly fuzzing techniques and their combinations. Fuzz testing or fuzzing was first proposed by Miller et al. in 1990 [83] to understand the reliability of UNIX tools. This is known as black-box fuzzing technique because it does not require any structural information of the PUT; in fact the PUT is viewed as a black box. Basically, black-box fuzzing randomly mutates selected program inputs (i.e., modify part(s) of them) to produce other inputs which hopefully can trigger some abnormal behaviors of the program like crashes and hangs. Due to its lack of program understanding, inputs generated by black-box fuzzing are likely to be rejected by input sanitizer component (e.g., parser code). To address the problem, the work of [50] proposed white-box fuzzing technique (a.k.a concolic execution) that runs PUT both symbolically and concretely. This work explores into how PUT is running (e.g., control flows, data flows, CPU and memory states), and traverses deep paths that are unlikely to be reached by black-box approach. Although white-box fuzzing is more systematic, its scalability is limited due

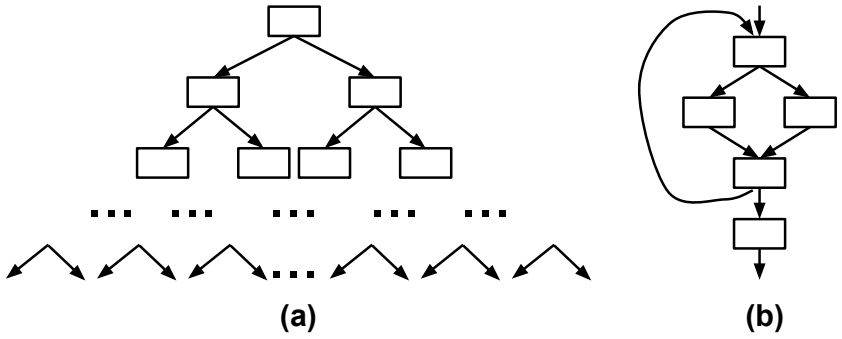


Figure 1-1: Control flow graphs of (a) a program having multiple branches and (b) a program with loop

to the well-known path explosion problem. In a program having multiple conditional branches and/or unbounded loops (as depicted in Figure 1-1), the number of program paths can grow exponentially.

Many research works have been done to maximize the effectiveness and efficiency for both black-box and white-box fuzzing techniques by intelligently control their search space. For instance, model-based black-box fuzzing [9, 11] and grammar-based white-box fuzzing [49] exploit input data model and input grammar to add more constraints to input data and hence prevent fuzzing engines from generating totally invalid inputs. BuzzFuzz [45] was designed to use taint analysis to identify parts of the input that control critical locations (e.g., system calls) and sensitive data so they focus modifying these parts instead of treating all parts equally. TaintScope [101] moved one step further to handle checksum integrity checks in highly-structured inputs. ESD [107] and KATCH [81] systematically direct symbolic exploration towards some specific location using control flow graph extracted from program source code. In addition, several powerful symbolic execution frameworks supporting smart search heuristics and ability to work with real-world (binary) applications have been developed such as BitBlaze, SAGE, Mayhem, KLEE and S2E [51, 28, 95, 35, 33], to name a few.

Coverage based grey-box fuzzing (grey-box fuzzing for short) [4, 5, 8] has gained lots of attention from both academia and industry. Unlike black-box fuzzing, grey-box fuzzing has access to some internal program structure like branch coverage information. Unlike white-box fuzzing, grey-box fuzzing uses lightweight instrumentation to determine control flow coverage – no heavy program analysis and no control flow graph extraction are needed. The technique turns out to be effective and efficient in finding bugs, especially security related ones. Recently, Stephens et al. [96] have combined grey-box fuzzing and white-box fuzzing in a framework named Driller to leverage the best of both worlds.

All of the aforementioned research has significantly improved fuzzing techniques; however, they still have several drawbacks. Grammar based white-box fuzzing only works with context free grammar and cannot handle integrity checks which are very common in highly-structured file format like PNG, WAV and PDF. The directed search algorithms in ESD and KATCH are smart but they work on subjects where source code is available, and hence more precise structural information is available. Taint analysis directed fuzzing approaches like BuzzFuzz and TaintScope implicitly assume they have seed inputs to reach critical functions. It means they ignore handling reachability analysis which is extremely challenging in program binaries. Meanwhile, coverage-based grey-box fuzzing is still undirected – given a specific target location, it cannot be directed towards quickly generating seeds that can reach the target.

Due to the pointed limitations, given an ***inadequate test suite***, the current fuzzing techniques are not skillful at directing the exploration to reach ***certain locations*** and expose errors in ***large program binaries*** that take ***highly-structured*** file inputs.

1.1 Thesis Overview

In this thesis, we propose techniques to enhance the directed search algorithms in major types of fuzzing – black-box, grey-box and white-box fuzzing. Our algorithms take into account the *inadequacy* of given test suite, the *complex structures* of program inputs (e.g., the presence of optional data chunks, integrity checks like checksum), the *incompleteness* of program structure (e.g., control flow graph) lifted from binaries, and also the *complexity* of the program under test (e.g., multi-module design). Our final goal is to develop a fuzzing-based automated testing framework that scales well to large real-world (binary) programs. Moreover, being aware of the overwhelming number of failing tests could be generated during fuzzing process, we also develop a fine-grained bucketing technique to effectively manage and group the tests to ease the debugging phase.

To this end, we first design a directed search algorithm in Hercules, a symbolic execution based white-box fuzzing that works directly on (stripped) program binaries. Given a target location l and a set of benign inputs T in which none of them reaches l, Hercules can generate concrete test case(s) to reach l and satisfy given certain condition (e.g., crash condition). The directedness of Hercules is attributed to its ability to bound the exploration space using the module dependency graph and control flow graph lifted directly from application binaries. Moreover, once Hercules reaches location l but the given condition is not satisfiable, it leverages the minimal unsatisfiable core produced by SMT constraint solver to first extract the reason of contradiction (in form of branch condition(s)). Afterwards, based on the reason of the contradiction Hercules can identify which conditional IF statement caused the contradiction and hence backtrack to it, negate the branch and follow the

opposite program path towards location l.

Hercules requires a satisfactory test suite having benign test input to explore paths towards some control location which is close enough to the target location l. For example, the control location can be an entry point of the module (e.g., shared library) that contains l. However, the assumption may not always work in programs taking highly-structured file formats such as PNG, WAV and PDF because in these file formats, there are several optional data chunks which normally do not exist in an inadequate test suite. In fact, the existence of optional data chunk(s) at certain place(s) in a file input can decide whether a control location of the consuming programs can be reached. To tackle the problem, we propose MoBWF, a novel combination of model-based black-box fuzzing (as embodied in Peach Fuzzer[9]) and directed white-box fuzzing (as implemented in Hercules) to exploit the best of both worlds. We implement in Peach a so-called "data chunk transplantation" capability to generate and insert missing data chunk under guidance of Hercules while Hercules effectively explores the value space of data bytes in the newly inserted chunk to drive the search towards the target location.

MoBWF significantly improves effectiveness and efficiency of model-based black-box fuzzing and white-box fuzzing in large program binaries taking highly-structured file inputs. Nevertheless, it is still a heavyweight technique because it involves complicated program analysis and constraint solving. In contrast, coverage-based grey-box fuzzing (CGF) is a lightweight technique that has impressive records in discovering (security) bugs. However, its limitation is the lack of directedness. Recently, Böhme et al. [5] have found that CGF can be modeled as Markov chain. The finding opens a chance to integrate Markov Chain Monte Carlo meta-heuristics into CGF and make it

directed. To realize the observation, we build AFLGo – a directed CGF by integrating into it the Simulated Annealing [65] – a Markov Chain Monte Carlo approach. Simulated annealing allows AFLGo to regulate the so-called "fuzzing energy", which decides how much time should be allocated to fuzzing a selected seed, for a seed input based on its path distance to provided target location(s) and the current "temperature". The reader can refer to Section 6.4.1 for a full definition of path distance. The novel combination allows AFLGo to focus on mutating more "interesting" seeds but still take into account less "interesting" seeds and hence increase the chance to discover program bugs – which is equivalent to the *global* optimum in original Simulated Annealing.

A fuzzing tool could generate an overwhelming number of failing test cases where many of the tests are likely to fail due to same "reason" and hence it wastes developers' time and efforts. So bucketing – a technique to effectively group the failing tests – would be extremely useful. So far, people have only used pretty coarse grained run-time information like point-of-failure or stack trace to do the grouping. In this thesis, we propose a new symbolic analysis-based clustering algorithm that uses a semantic "reason" behind failures to group failing tests into "meaningful" clusters. The reason is defined as the constraint introduced by the branch at which the failing execution path deviates from the nearest correct path. The experimental results show that our technique is effective at producing more fine grained clusters as compared to the point-of-failure based and call-stack-based clustering schemes. As a side-effect, our technique also provides a semantic characterization of the fault represented by each cluster – a precious hint to guide debugging.

The main contributions of this thesis are as follows.

- **Unsat-core guided search algorithm in white-box fuzzing**.

The search algorithm in Hercules detects *reasons for infeasibility* of non-crashing paths and directs concolic execution towards target location until the location is reached and the crash condition is satisfied. The algorithm is supported by program structure information (e.g., module dependency graph and control flow graph) lifted directly from program binaries and several heuristics working on loops and string manipulation functions.

- **Leveraging input model to handle inadequate test suite of highly-structured input files**. Input model is used to glue model-based black-box fuzzing (as embodied by Peach Fuzzer [9]) and directed white-box fuzzing (as embodied by Hercules) together to handle missing data chunk problem due to inadequate test suite. We design a so-called "data chunk transplantation" technique that can "transplant" missing data chunk(s) into an input under the guidance of white-box fuzzing and some input model. While white-box fuzzing informs Peach about what and where to transplant, input model helps Peach to decide how to transplant.

- **Directed coverage-based grey-box fuzzing**. We integrate Simulated Annealing – a Markov Chain Monte Carlo approach into coverage-based grey-box fuzzing to direct the exploration towards a given set of target locations. To the best of our knowledge, we develop the first multiple-target search-based software testing technique where the single objective is to generate an input that exercises as many of the given targets as possible.

- **Fuzzing framework and evaluation**. We have developed three fuzzing systems, which are Hercules, MoBWF and AFLGo, and evaluated them on two main applications – patch testing for

vulnerability detection and crash reproduction. These toolset successfully reproduce crashes in large real-world (binary) programs (e.g., Adobe Reader, Windows Media Player, OpenSSL, Binutils etc) taking highly-structured file formats (e.g., PDF, PNG, WAV etc). Notably, AFLGo can expose the well-known HeartBlead vulnerability in OpenSSL library almost four (4) times faster than the state-of-the-art AFL fuzzer. AFLGo has discovered 14 zero-day vulnerabilities in Binutils' utilities; all of them have been confirmed and fixed by Binutils' maintainers. Five (5) CVEs have been assigned to critical vulnerabilities.

- **Fine-grained failing tests bucketing technique**. We leverage symbolic analysis and symbolic execution tree to identify semantic "reasons" behind failures and group failing tests into "meaningful" clusters. The semantic reason makes our approach more fine-grained compared to off-the-self point-of-failure and call-stack-based approaches.

Impact on current state of practice. Our proposed techniques can be applied to various applications; some applications will be shown in this thesis such as crash reproduction of field-failures, vulnerability detection, patch testing and debugging. Using our techniques, developers can do fuzz testing on large real-world (binary) programs taking highly-structured inputs in the presence of inadequate test suites. Our crash reproduction and vulnerability detection techniques need minimal to no help from developers – all information can be extracted automatically from software version in patch testing or little information about call stack needs to be provided. Our bucketing method can significantly reduce number of failing tests to be analyzed and hence save substantial time and efforts of developers.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. We first discuss about prerequisite knowledge and related work in Chapter 2 and Chapter 3. In Chapter 4, we present our directed search algorithm for white-box fuzzing to reach target location. Chapter 5 shows the combination of model-based black-box fuzzing and directed white-box fuzzing. In Chapter 6, we describe how we make coverage based grey-box fuzzing directed. Chapter 7 shows the symbolic analysis based bucketing technique and Chapter 8 concludes this thesis as well as discuss about potential future research directions.

Chapter 2

Background

Based on the awareness about structure of the program under test we can categorize fuzz testing techniques into three main types – black-box, grey-box and white-box fuzzing. While black-box technique has no information about program structure, the white-box one has access to both control flows and data flows of the PUT. Grey-box fuzzing lies in between, it might have some (partial) information about program control flows (e.g., branch coverage) or about data flows (e.g., via taint analysis). In this chapter we explain how different fuzz testing techniques work in details and indicate their advantages and disadvantages in software testing.

2.1 Running Example

To make a clear explanation and comparison between black-box, grey-box and white-box fuzzing, we design an example which could challenge all the three techniques. Listing 2.1 shows the example in which a simple structured file is read. As depicted in Figure 2-1, the file starts with a signature (a.k.a a magic number) having value of 0x41424344 in hexadecimal or "ABCD" string in ASCII followed by the number of data

blocks (a.k.a data chunks), and all data blocks. Each data block start with its data size followed by the block's data and a checksum value calculated on the data for integrity assessment. This simple file format resembles several widely-used files such as PNG, PDF and WAV. The program will crash (at line 16 in Listing 2.1) if the file is valid – its signature and the checksum values calculated on blocks' data are correct – and the value returned by processing content of some block's data equals to a specific value (line 15).

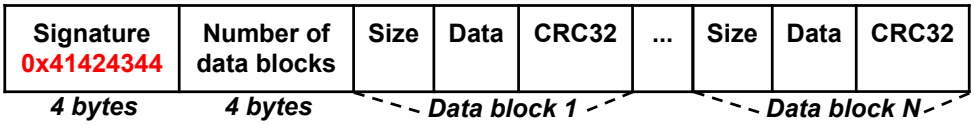


Figure 2-1: Example file format

```
1  int main(void)
2  {
3    file_t *file = read_file();
4    if (file->signature != 0x41424344) {
5      puts(``Unsupported file format'');
6      return 1;
7    }
8    for (int i = 0; i < file->block_size; i++) {
9      block_t *block= get_next_block(file);
10     if (block->crc32 != calculate_crc32(block->data
         )) {
11       puts(``Bad checksum value'');
12       return 2;
13     }
14     int result = process_block_data(block->data);
15     if (result == CRASH_VALUE) {
16       program_crash();
17     }
18   }
19   return 0;
20 }
```

Listing 2.1: Example program which processes a structured file

2.2 Black-box Fuzzing

Miller et al. pioneered the idea of fuzzing in 1991 [83] to examine the reliability of UNIX command line utilities. The first fuzzing technique is known as Black-box Fuzzing because no structural information of the program under test is required. It randomly mutates/modifies selected program inputs (i.e., seed inputs) using several mutation operators (e.g., bit-flip, boundary value substitution, block deletion and duplication) to generate massive number of new inputs before feeding them to PUT. Then, the PUT is executed and monitored to capture abnormal behaviors like program crashes or hangs. The effective yet simple technique paved the way for later research on automated software testing. However, since the traditional black-box fuzzing technique (as embodied by zzuf[76]) does not take into account the structures of program inputs (i.e., file formats or network protocols), it is very likely that a large portion of the generated inputs are rejected by the PUT's parser since they do not conform to the structure of expected input. The early rejection prevents traditional black-box fuzzing from digging into deep paths of the program to find persistent bugs.

Refer the the running example in Listing 2.1, a traditional black-box fuzzing could easily get stuck at generating some input having correct signature to bypass the very first check (at line 4). In fact, the chance for a single try to randomly generate a specific 32-bit value is extremely slim, just one out of four billions. If an adequate corpus of valid benign seed inputs is provided, the fuzzer should be able to pass the first check and continue mutating the main contents of the seeds. However, the inputs generated by mutating a valid seed still have almost no chance to trigger the crash because of the checksum integrity check at line 10.

```
<DataModel name="FileModel">
    <Number name="Sign" size="32" value="0x41424344
       "/>
    <Number name="BlockSize" size="32">
      <Number size="32" signed="false">
      <Relation type="count" of="Blocks" />
  </Number>
    </Number>
    <Block name="Blocks" maxOccurs="10000">
      <Number name="Size" size="32"/>
      <Blob name="Data"/>
      <Number name="Crc32" size="32">
        <Fixup class="Crc32Fixup">
            <Param name="ref" value="Data" />
        </Fixup>
    </Number>
    </Block>
</DataModel>
```

Listing 2.2: Peach data model for the sample file format

To address this problem, Model-based Black-box Fuzzing technique was proposed and implemented in several fuzzing frameworks like Peach and Spike[9, 11]. Essentially, the technique leverages information of input's structure to mutate program input (e.g., input file or network message) in a smarter way; it only modifies "mutable" part(s) of the input and makes the whole input valid "enough" respect to the defined structure. The technique improves the validity of generated inputs and hence it is more likely to reach deeper and critical program statements before exposing program bugs. Listing 2.2 presents how the sample structured file depicted in Figure 2-1 can be modeled using Peach modeling language. The Peach modeling language allows to specify a file format as Peach Pit [10]. It uses primitive data types (e.g., number and flag) and composite ones (e.g., string, block, blob) to describes data blocks and data fields. Moreover, it can model the relationships (e.g., size-of, count-of, offset-of) between data blocks and data fields. Peach also supports "fixups" and "transformers". While fixups allow to repair related data fields like checksums, transformers

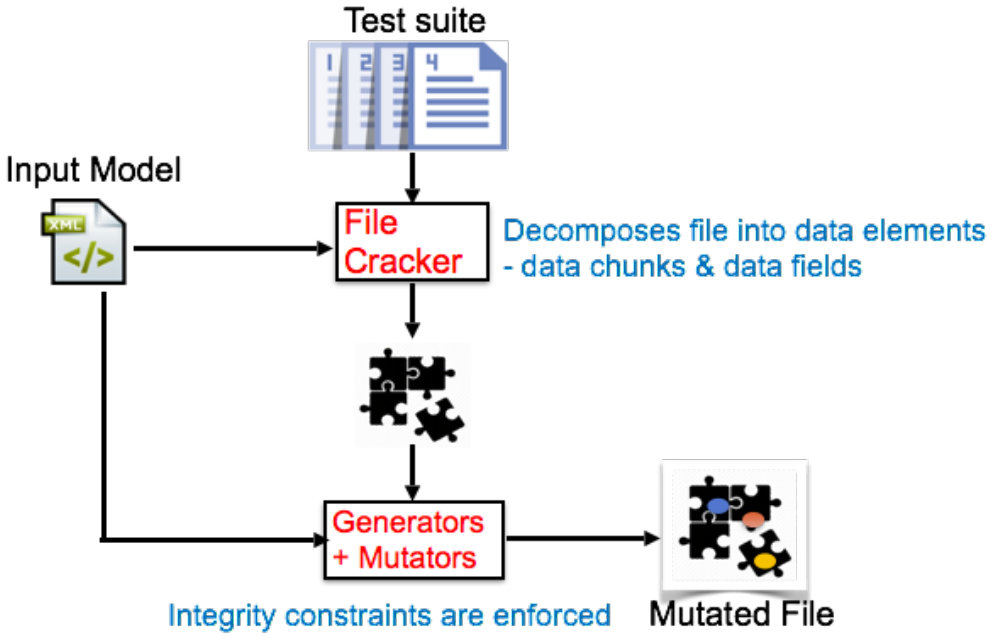


Figure 2-2: Architecture of Peach as a File Fuzzer

are used for encoding, decoding and compression.

Figure 2-2 shows the architecture of Peach fuzzer. One unique component of Peach is the Cracker. Given a set of seed inputs and the written input model, Peach uses Cracker to decompose each seed input into smaller data parts based on the input format specified in the model. Once the seed is decomposed, Peach applies several mutation strategies to modify data parts before reassembling them to create new complete inputs and sending the inputs to PUT. All data parts are mutated with the support of input model so the generated inputs are valid respect to the model. As a result, the inputs are more likely to be accepted by the parser code in the PUT and they could lead to deeper paths to expose hidden program bugs. Specifically, in the example code in Listing 2.1, Peach would easily generate inputs to reach the processing code part (lines 14-17) and it could generate some data block to satisfy the crash condition to trigger program crash (line 15).

In several cases generating such crashing data block would challenge even the model-based black-box fuzzers like Peach. Peach would be good at generating data fields that have some boundary values (i.e. maximum and minimum of a data type); however due to its randomness, it is difficult to produce some specific value especially in case the values are calculated on several distinct data fields. Moreover, Peach requires input models which could be tough to construct especially in case of proprietary file formats or protocols – no specification is available.

2.3 White-box Fuzzing

2.3.1 Symbolic Execution

Symbolic execution is a powerful program analysis technique invented by King in 1976 [63]. Even though it was introduced long time ago, the technique has only been widely used in software testing recently because of the advancements in computer architecture and constraint solver. Unlike normal program execution (a.k.a concrete execution), symbolic execution does not take concrete values as inputs. Instead, it starts with symbolic inputs (i.e., the inputs can take any value in the value ranges of corresponding data types) and (theoretically) explores all feasible program paths. To this end, at each conditional IF statement it collects condition(s) of feasible branch(es), namely branch condition(s), and encodes the condition(s) in logical formula(s). The conjunction of all branch conditions along a specific program path π forms path constraint/path condition which captures the set of all inputs executing π. The feasibility of a path is decided by invoking a constraint solver like Z3 [40].

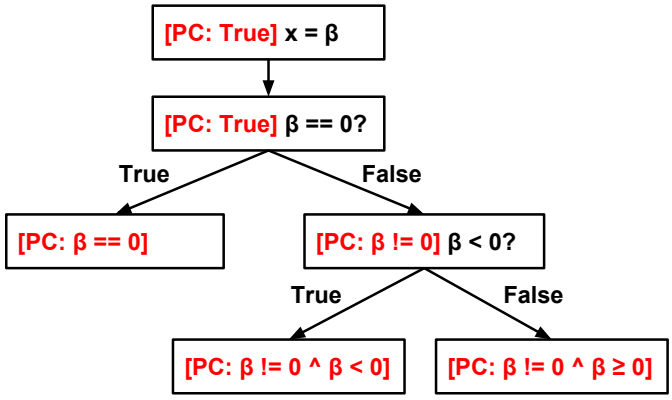


Figure 2-3: Symbolic execution tree of get_sign program

```
1  int get_sign(int x)
2  {
3    if (x == 0) {
4      return 0;
5    } else if (x <0 ) {
6      return -1;
7    } else {
8      return 1;
9    }
10 }
```

Listing 2.3: Example program get sign of an integer number

Listing 2.3 shows a simple example we use to demonstrate how symbolic execution works. The program takes an integer number as input and returns a value representing the sign of the input. It has three distinct paths where x equals to zero, x is smaller than zero and x is bigger than zero. Figure 2-3 is the so-called symbolic execution tree of the example program. Symbolic execution starts by substituting the concrete program input (variable x in the example) by a symbolic value β. At each program statement, symbolic execution maintains a program state called symbolic state; a symbolic state keeps the current path condition, which is the conjunction of all preceding branch conditions and a symbolic memory state that manages the symbolic variables and its propagation during program execution. The initial symbolic state is created when the

program starts and the path condition is set to true. When a conditional statement is executed, symbolic execution requests the integrated constraint solver to check the feasibilities of both two branches. For each feasible branch, it updates the symbolic state (i.e., path condition and memory state) and follows the branch to explore deeper code part(s). For example, at line 3 where the current path condition PC is still true, symbolic execution checks whether β can be zero. It is obvious that both the two branches (β equals to zero and β is different from zero) are feasible so two new symbolic states are created with updated path conditions and memory states – one for the True branch and one for the False branch. Similarly, two symbolic states are created for the IF statement at line 5. As a result, all three paths of the example program have been explored, and the constraint solver can be invoked to generate three concrete values of the symbolic variable β for these paths accordingly.

2.3.2 Symbolic Execution based White-box Fuzzing

Symbolic execution is a systematic program analysis technique and it can complement the randomness of black-box (and even grey-box) fuzz testing. In 2005, [50] and [30] independently proposed ideas to leverage symbolic execution for systematic testing by running symbolic execution along with concrete execution. The technique is now known as Symbolic Execution based White-box Fuzzing (white-box fuzzing for short) or concolic execution. In DART [50], it first runs the test program on one random input and symbolically gathers constraints at decision points (e.g., conditional IF statements) that use input values. Then, DART negates these symbolic constraints one by one to generate new test cases. The process is repeated so that DART can explore large number of

program paths deviating from the path followed by the selected random input. In EGT [30], rather than running the test code with concrete input which can be manually written or generated by random testing, EGT runs it on symbolic input that is initially allowed to be "anything". As the code executes and processes the input, at each branch point EGT "fork" the execution for feasible path(s) – create new symbolic state(s) and update the path condition(s). Once a concrete value is needed (e.g., to interact with outside network interface and external libraries or to get a concrete test case) EGT uses constraint solver to solve the corresponding path constraint.

Using symbolic analysis and constraint solver, both DART and EGT are effective at reasoning about specific values. As a result, they can easily generate input to pass the first check (magic number check at line 4) and the crash condition (the check at line 15) of the running example at Listing 2.1 which are challenging for (model-based) black-box fuzzing. However, they would get stuck at reasoning about the loop condition (at line 8) and the checksum validation (at line 10) due to the path explosion problem. That is, in large programs or even in small programs having several conditional branches and/or loops, the number of program paths can grow exponentially increase so that the computing resources can quickly get exhausted. Several studies have been conducted to tackle the problem, we will discuss them in details in Chapter 3.

In more than ten years, since 2005, several white-box fuzzing tools have been developed based on the core ideas of DART and EGT such as SAGE, KLEE, BitBlaze, S2E, Mayhem [51, 28, 95, 35, 33], to name a few, some are closed-source (SAGE, Mayhem) while others are open access. These tools have shown successful applications in both industry and

academia. Very recently, Microsoft have released its Springfield[1] fuzzing-as-a-service project in which SAGE is a key component. In this thesis, we have extensively used two open-source symbolic execution engines KLEE and S2E for implementing our proposed ideas. Thus we discuss here the architectures of the tools in more details to ease the understanding of our implementations.

Cadar et al [28] first introduced KLEE at the OSDI conference in 2008. KLEE is implemented as a virtual machine working on LLVM bytecode [67] instead of machine code. The original version of KLEE supported LLVM-2.9, and now in its newest version KLEE already supports LLVM-3.4. KLEE takes LLVM bytecode as input, executes the bytecode to explore different program paths, and uses run-time checkers (e.g., memory access violation checker) to detect program bugs and generate test cases for both benign and buggy paths. KLEE has shown its ability to automatically generate test inputs to get high code coverage and find deep bugs in the Coreutils and BusyBox toolset.

Figure 2-4 shows the architecture of KLEE. Like other symbolic execution engines, a major component in KLEE is the interface with constraint solver. In current version, KLEE supports metaSMT interface to provide a handful of choices including STP, Boolector and Z3 solver. Another important component of KLEE is its set of search strategies. KLEE provides a rich set of search strategies for different objectives – random search, explore deeply (Depth first search), explore widely (Breadth first search), maximize code coverage and the interleaving between them. It also supports a clean interface for developing new search strategy for specific requirements.

[1]Springfield service: https://www.microsoft.com/en-us/springfield/

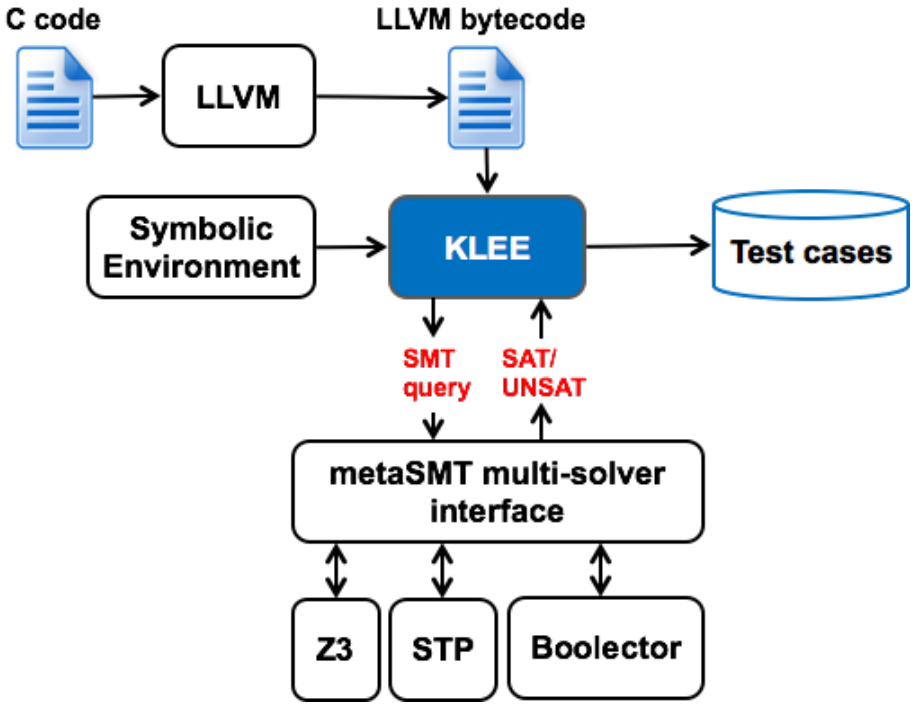


Figure 2-4: Architecture of KLEE

```
1 void main() {
2   uint32 x,y;
3   fread(stdin, &x, sizeof(x));
4   make_symbolic(&x, sizeof(x));
5   make_symbolic(&y, sizeof(y));
6   y = f_env(x);
7   if(y == 0) abort;
8 }
```

Listing 2.4: Environment interaction in KLEE

In Figure 2-4, one notable component of KLEE is symbolic environment. The advantages of the component are threefold. First, it allows KLEE to mitigate path explosion problem using simplified versions of external libraries (e.g., standard libc). Second, KLEE can model closed-source libraries based on their provided/inferred specifications to symbolically execute programs working with these libraries. Third, KLEE can also model specific environments like network communications [92] to test network applications. Despite the advantages, the environment

modeling is somewhat adhoc and manual. Listing 2.4 shows a contrived example in which the program calls a function, namely **f_env** and we have no access to its source code. Based on a specification of the function, we need to manually write a simplified version of **f_env** and compile it to LLVM bytecode to make KLEE work. KLEE runs and generate a test case that triggers the crash at line 7. However, it is possible that we cannot reproduce the crash on the binary version of *f_env* because if in this original version of *f_env* there is no paths to output the value of 0 to satisfy the crash condition. It means KLEE raises a false alarm. Therefore, a (semi) automated environment modeling approach is in high demand and it is still an open research problem.

In 2011, Chipounov et al [35] introduced S2E, a new symbolic execution framework which can directly work on program binaries. Moreover, S2E supports symbolic execution in full software stack (i.e., from application down to C-library, OS kernel and device drivers) and hence unlike KLEE, S2E requires no abstractions (i.e., models) for the operating system and external libraries. As a result, in our example listed in Listing 2.4, S2E should not report any error because there is no feasible path to trigger the crash. S2E is implemented by augmenting QEMU emulator [98], specifically the dynamic binary translator, to make it work with KLEE. The modified dynamic binary translator can translate binary code into LLVM bytecode which can be interpreted by KLEE.

To mitigate the path explosion problem, S2E supports Selective Symbolic Execution to choose which modules to be run in symbolic mode and leave other modules run in concrete mode. More specifically, while running in symbolic mode, S2E extracts and converts the current Translation Block (i.e., a block of binary code) into LLVM bytecode and pass it to KLEE. While running in concrete mode, S2E runs similar to

QEMU. S2E needs to switch back and forth between symbolic mode and concrete mode, so it has to maintain the consistency of the whole system execution.

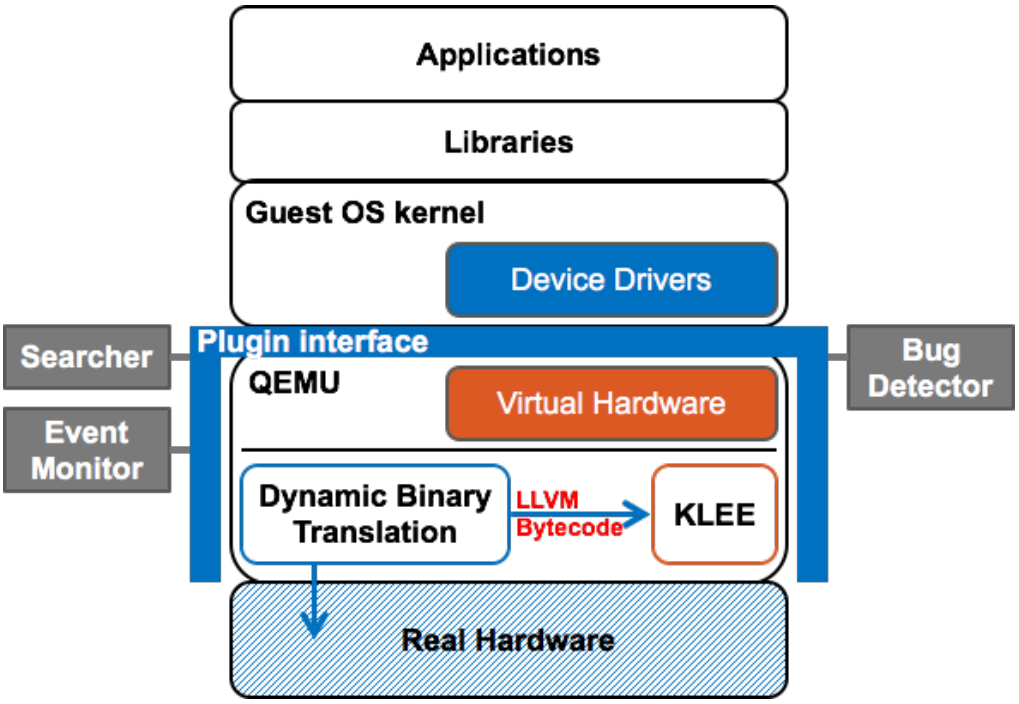


Figure 2-5: Architecture of S2E

The architecture of S2E is shown in Figure 2-5. It is designed in such a way that users can easily extend its functionality. Apart from the core of S2E, other components can be implemented as plugins to the core. A plugin can belong to the Selector or Analyzer category depends on its functionality. A selector plugin selects state to be run or choose which modules to be symbolically executed. For instance, a search algorithm can be implemented as a selector plugin. An analyzer plugin is used to analyze the execution information; some examples are the memory analyzer, memory checker, crash detector.

2.4 Coverage-based Grey-box Fuzzing

Coverage-based grey-box fuzzing (CGF) [4, 5, 8] leverages control flow information of the program under test to guide random testing. For instance, AFL fuzzer [4], the state-of-the-art CGF fuzzing tool, has been widely used in industry and academia to find remarkable number of security vulnerabilities in real-world programs. To achieve the effectiveness, AFL relies on lightweight instrumentation mechanism which allows it to capture basic block transitions and coarse branch-taken hit counts information in run-time.

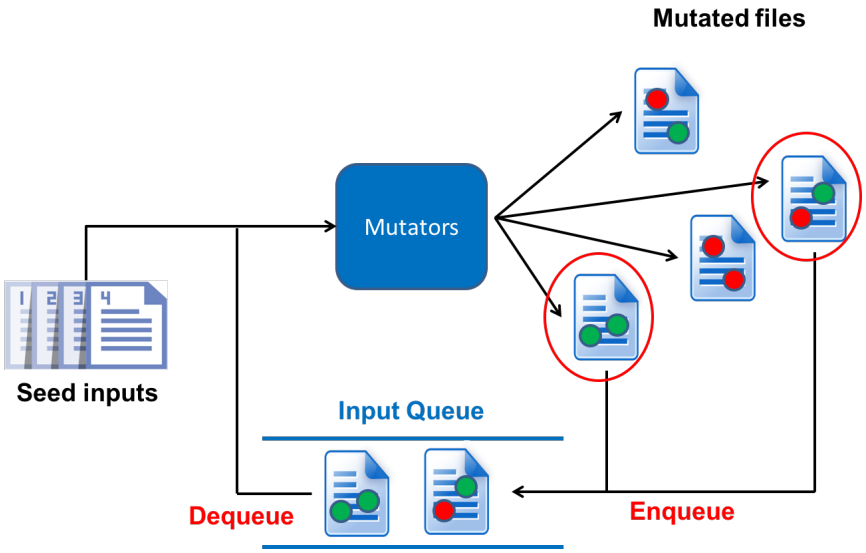


Figure 2-6: Architecture of AFL as a File Fuzzer

Figure 2-6 displays the architecture of AFL as a file fuzzer. Note that AFL can be used to test command line and network based applications as well provided that some adapter needs to be designed to convert back and forth between file and command arguments or network messages. Since AFL has information about the edge coverage of the mutated inputs, it can decide which one is interesting (e.g., covers new edge(s)) and should be retained for further fuzzing phases. AFL maintains an input queue which only keeps interesting inputs. For each fuzzing round, it iterates

through items in the input queue, mutates them and enqueues inputs that trigger new behaviors. Since the queue is getting larger, AFL has smart heuristics to rank items in the queue so that those have higher rank should be prioritized. Meanwhile, the lower ranked items can be skipped with some probability. The input queue is designed to be easily shared between different fuzzing engines. This feature allows us to run AFL in parallel mode or run AFL with other test generation engines.

```
1 cur_location = <COMPILE_TIME_RANDOM>;
2 map_index = cur_location ^ prev_location
3 shared_mem[map_index]++;
4 prev_location = cur_location >> 1;
```

Listing 2.5: AFL's instrumentation.

The instrumentation can be done during compilation time or directly on program binaries. Listing 2.5 shows the pseudo code for instrumentation in AFL. AFL allocates a shared memory region – **shared_mem** – to keep track the hit counts. At each control statement (e.g., conditional jump or call), AFL randomly generates a value for the *cur_location* variable and uses this value with its previous value to calculate the index to the shared memory. The index calculation is performed in such a way that AFL can distinguish between two basic block transitions – (A,B) versus (B,A).

In the running example shown in Listing 2.1, CGF would be more effective than both black-box fuzzing and white-box fuzzing on handling the loop (at line 8). Indeed, CGF only retains inputs covering new control-flow behaviors (e.g., new branch or new number of loop iterations), so it does not waste time to generate huge number of inputs which might cover same behaviors like black-box approach. In addition, since CGF only keeps the control flow transitions of "interesting" inputs,

not the full program state for each program path (i.e., registers' state, memory state and path condition) like symbolic-execution based white-box fuzzing, CGF can easily explore the loop with no scalability problem. However, like black-box fuzzing, CGF could get stuck at generating specific values for the file signature (line 4) and the crash condition (line 15). Moreover, it has almost no chance to bypass the checksum validation (line 10). In practice, one normally tries to identify the checksum check in source code or in binary and disable it or jump out of it. Once the crashing input is generated, the checksum will be repaired [101].

Chapter 3

Literature Review

In this chapter we present related work in improving the effectiveness and efficiency of fuzz testing techniques. The improvements mostly come from 1) techniques to enhance the directedness of fuzz testing, 2) techniques to tackle the path explosion problem of symbolic execution and 3) the combination of different fuzzing approaches to leverage the best of all. Due to the advances in fuzz testing techniques, they have been used broadly in industry and academia to discover program defects. A common problem is that the fuzzing tools normally generate an overwhelming number of failing test cases where many of the tests are likely to fail due to same "reason". So in this chapter, we also review relevant research on bucketing techniques which aim to group similar failing tests together to significantly reduce number of tests to be analyzed. The results of bucketing techniques would remarkably ease the debugging process.

3.1 Enhancing Directedness in Fuzz Testing

Since the search space to explore a real-world program could be huge, a rich set of approaches has been proposed to make fuzzing techniques more

targeted [46, 53, 101, 81, 107, 31, 87, 93]. That is, the approaches focus on exploring "critical" program locations (e.g., system calls, memory access instructions, code changes etc) instead of treating all code parts equally.

BuzzFuzz [46], TaintScope [101], Flax [93] and Dowser [53] leverage the taint analysis to localize the program input should get be stress tested. Basically, they first run the test program with some input and use taint analysis to locate which part(s) of the input can control critical program locations. Afterwards, they spend much more effort to "mutate" the located the input part(s) to discover program bugs. The mutation can be done in black-box manner using mutation operators or in white-box way using constraint solver. TaintScope [101] can automatically identify integrity checks (e.g., checksums) in program binaries and bypass the checks. Flax [93] uses dynamic tainting on client web applications and directed fuzzing to discover client-side validation (CSV) vulnerabilities. *Dowser* uses taint analysis to identify program inputs that influence memory accesses and uses concolic execution with partially symbolic inputs for learning about pointer access patterns [53]. Using this information, *Dowser* steers fuzzing technique towards complex pointer calculations in the program.

Debugging approaches such as ESD [107], BugRedux [58] and patch testing approach KATCH [81] systematically direct symbolic exploration towards a specific program location. ESD and KATCH utilize the program source code to extract system information including inter-procedural CFG to inform the techniques and apply data flow analysis. Both approaches analyze data-flows to identify reaching definitions responsible for taking critical control-dependent edges and steer symbolic execution towards these intermediate goals using proximity metric. BugRedux also requires source code because it needs to instruments program under test before deploying

it to the user side. Once a field failure happens, all execution data (e.g., point of failure, stack trace etc) is collected. First, BugRedux uses the execution data from the field to identify a set of intermediate goals that can guide the exploration of the solution space. Second, it uses a heuristic based on distance to select which states to consider first when trying to reach an intermediate goal during the exploration.

Approaches that work on program binaries focus on resolving sufficient system information using static and dynamic analyses [17, 36, 32]. Approach by Babić et al. uses static analysis to guide automated dynamic test generation [17]. Dynamic analysis resolves indirect jumps with seed tests, and the static analysis helps symbolic execution directing exploration towards vulnerabilities based on the shortest paths and loop pattern heuristics. *MACE* by Cho et al. combines symbolic and concrete execution to build and refine an abstract finite state model of the system-environment interaction and use it to guide the program exploration [36]. *HI-CFG* by Caselden et al. generates hybrid information- and control-flow graph of a program to direct stages of backwards symbolic execution. Brumley et al. [26] lift control flow graph from the intermediate representation of the program and compute a "chop" of the graph which includes only those program paths which may reach the vulnerability point. The approach prunes paths that might not be relevant to reaching the target location.

In summary, to make fuzz testing more directed previous research requires source code to extract precise program structure. Some can work directly on program binaries but cannot handle highly-structured inputs or assume the availability of adequate test inputs to reach critical locations. In Chapter 4 and Chapter 5, we show that our directed white-box fuzzing approach and its closed-loop combination with

model-based black-box fuzzing work directly on (stripped) program binaries in the presence of inadequate test suite of highly-structured file formats. In Chapter 6, we present directed coverage-based grey-box fuzzing in which no taint analysis is involved.

3.2 Improving Scalability of Symbolic Execution

Path explosion is the main problem which limits the scalability of symbolic execution. A lot of research has been done to tackle and mitigate the problem.

Loops and string manipulation functions are two main causes of path explosion. Many techniques have been proposed to handle loops and strings in symbolic execution [68, 27, 106]. Larson and Austin characterize and track bounds and null termination of string variables for dynamically checking validity of program inputs [68]. Bucur et al. associate high-level execution paths of the program to some low-level execution paths during symbolic execution of *python* programs [27]. Xie et al. [106] propose a classification of multi-path loops to understand the complexity of the loop execution and use a path dependency automaton (PDA) to capture the execution dependency between the paths.

The idea of summarizing functions or problematic behavior in symbolic execution have been investigated earlier. Godefroid proposed a compositional approach to capture and reuse function summaries to scale dynamic symbolic execution [48]. Brumley et al. describe vulnerability signatures as weakest preconditions [23]. Several approaches have explored similar intuition for summarizing and reasoning about problematic behavior in a backwards fashion to find program inputs that

trigger such behavior [25, 80, 37, 32].

Kuznetsov et al. introduced dynamic state merging and query count estimation [66]. By estimating the impact of symbolic variables on solver queries their approach merges states balancing between the number of generated states and the complexity of the queries to the solver. *Mayhem* by Cha et al. combines online and offline symbolic execution and models symbolic memory at the binary level [34]. Built on *Mayhem*, *Veritesting* enhances dynamic symbolic execution with static symbolic execution [16].

Grammar-based Whitebox Fuzzing (GWF) [49] generates inputs that are valid w.r.t. a context-free grammar $\mathcal{G}$. By that, it can prune paths leading to invalid inputs and hence reduce the exploration space. We use the example in Listing 3.1 for illustration.

```
1  int i;
2  char* input;
3  char getNextToken() {
4    return input[i++];
5  }
6  bool isSorted() {
7    int prev_digit = 0;
8    if ('{' == getNextToken()) {
9      do {
10       char token = getNextToken();
11       if (',' == token) continue;
12       if ('}' == token) return true;
13       int digit = asInt(token);
14       if (prev_digit > digit) return false;
15       prev_digit = digit;
16     } while (true);
17   }
18   return false;
19 }
```

Listing 3.1: isSorted() returns true if the input is a sorted list of single digit numbers

The context-free grammar $\mathcal{G}$ may be written as

$$\mathcal{G} \rightarrow \{Numbers\} \tag{3.1}$$
$$Numbers \rightarrow Numbers\texttt{,}Numbers \tag{3.2}$$
$$Numbers \rightarrow Digit \tag{3.3}$$
$$Digit \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \tag{3.4}$$

which encodes that valid inputs start with an open curly bracket followed by a comma-separated list of (at least one) digits and a closing curly bracket. GWF encodes a path condition as regular expression such that a context-free constraint solver can generate an input that is accepted by both, the grammar and the regular expression. For input {1,2}, GWF yields the following constraint $\mathcal{R}$ to explore the alternative branch where the input does not end in a curly bracket. Notice that to ease the explanation, we show $\mathcal{R}$ in a user-friendly representation; GWF has a customized representation called regular path constraint.

$$\text{token}_1 = \{ \tag{3.5}$$
$$\wedge\ \text{token}_2 = Digit \tag{3.6}$$
$$\wedge\ \text{token}_3 = \texttt{,} \tag{3.7}$$
$$\wedge\ \text{token}_4 = Digit \tag{3.8}$$
$$\wedge\ \text{token}_5 \neq \} \tag{3.9}$$

where *Digit* is a symbolic variable. Using a context-free constraint solver, it is possible to derive an array with three digits that is accepted by both $\mathcal{G}$ and $\mathcal{R}$ (e.g., {0,0,0}). However, since the regular expression cannot express the arithmetic relationship between token_2 and token_4 (i.e., $1 < 2$), a completely different path might be exercised. This renders GWF both unsound and incomplete. Moreover, the context-free language which encodes the file format cannot express integrity constraints such as the checksum or the size of a data chunk. Functions computed over the data in a data field, such as a compression algorithm, cannot be expressed either.

In summary, previous techniques to tackle path explosion problem vary from function summary, path merging, loop analysis to grammar based white-box fuzzing. In Chapter 4 and Chapter 5, we will present our practical techniques to bound the loop iterations and group high-level paths of string manipulation functions to manage the increase of execution paths. We also leverage input models to prune most paths that are exercised by invalid inputs. Unlike grammar-based white-box fuzzing, our input models allow to specify integrity constraints and compression algorithms. Moreover, our technique maintains full path conditions as SMT formulas so it retains the soundness and completeness of symbolic execution.

3.3 Hybrid Fuzz Testing

Since no fuzzing technique is perfect, designing hybrid approaches to amplify the power and mitigate the weakness of each composing technique is a promising direction [86, 96].

[86] first proposed the idea of hybrid fuzz testing in 2012. In this approach, symbolic exploration is utilized to find "frontier nodes" and then fuzzing is invoked to execute the program with random inputs, which are preconstrained to follow the paths leading to a frontier node. Recently, Stephens et al. [96] released Driller – a hybrid fuzzing framework which combines the efficiency of coverage-based grey-box fuzzing and the effectiveness of symbolic execution based white-box fuzzing. White-box fuzzing is effective at reasoning about specific values while coverage-based grey-box fuzzing can efficiently explore program paths in a scalable way. In Driller, grey-box fuzzing initiates the fuzzing progress and it only seeks help from symbolic execution when it gets

stuck (i.e., grey-box fuzzing cannot discover new interesting paths) because it cannot generate some specific value (e.g., magic number). Symbolic execution takes a seed input from grey-box fuzzing, executes the program with the seed to collect path condition and negates constraints in the path condition to generate new inputs. Grey-box fuzzing takes some input which contains the required specific value and continues its fuzzing process. The hybrid approach provides an innovative way to leverage the advantages of both two worlds - grey-box and white-box fuzzing. However, the criteria to detect whether grey-box fuzzing is getting stuck is still very naive. Moreover, the power of symbolic execution is not fully exploited in this approach. Further research is needed to address these problems.

In Chapter 5, we present our approach to combine model-based black-box fuzzing and directed white-box fuzzing (MoBWF) to amplify the best of both worlds and target on programs taking highly-structured inputs. Driller does not primarily target such programs; in this respect, our approach is orthogonal to Driller. Driller can benefit from MoBWF when testing programs processing highly structured inputs.

3.4 Bucketing Failing Tests

One related line of research involves clustering crash reports or bug reports. This line of research is well-studied with several techniques have been proposed [39, 47, 62, 84, 91]. All of these techniques perform clustering based on the run-time information of programs. The Windows Error Reporting System (WER) [47] tries to place crash reports into various clusters using several heuristics involving module names, program versions, function offsets and other attributes. An improvement of the

bucketing in WER - ReBucket [39] - bases the clustering on specific attributes such as the call stack in an crash report. Crash Graph [62] uses graph theory (in particular, similarity between graphs), to detect duplicate reports. It builds a graph named *crash graph* for each crash report and detects the duplicate report by checking the similarity between two crash graphs. In terms of duplicate bug report detection, Runeson [91] proposed a technique based on natural language processing to check similarity of bug reports.

Another relevant work involves clustering program failing traces. Liu and Han [77] proposed the technique to use results of fault localization methods for clustering failing traces. Given two set of failing and passing traces which are collected from instrumented predicates of software program, they statistically localize the faults and two failing traces are considered to be similar if the pointed fault locations in the two traces are the same. Podelski et al.[89] cluster failure traces by building symbolic models of their execution (using model checking tools) and use interpolants as signatures for clustering tests. Due to the cost of symbolic model-checking, their technique seems to suffer from scalability issues as in their experiments, even their intra-procedural analysis times out (or the interpolant generator crashes) on a large number of methods.

Although the above-mentioned lines of research are relevant to our work, we target our research on clustering *failing tests* — instead of crash reports, bug reports or failing traces. Specifically, we work on failing tests obtained during symbolic exploration of software programs or provided by test teams.

To the best of our knowledge, all popular symbolic execution engines like KLEE, SAGE and MergePoint [29, 51, 16] only borrow and slightly change the techniques that have been proposed for clustering crash reports

to cluster their generated failing tests. The clustering approach can be as simple as using point of failure and error type in KLEE [29] or using call stack information in SAGE [51] and MergePoint [16].

Chapter 4

Directed Search in White-box Fuzzing

In this chapter, we present a directed method for generating inputs which reach a given "potentially crashing" location. Such potentially crashing locations can be found by a separate static analysis (or by gleaning crash reports submitted by internal / external users) and serve as the input to our method. The test input generated by our method serves as a witness of the crash.

4.1 Introduction

Complex software systems are released and deployed with faults. Some faults trigger application crashes that elevate system security risks and are difficult to trace, analyze and reproduce. The problem of finding crashing paths has been addressed by previous research, however, few techniques cope with large real-world binaries. Real-world binaries present challenges for program analysis techniques due to their size, complexity, and multitude and depths of execution paths. In addition, the information about structure

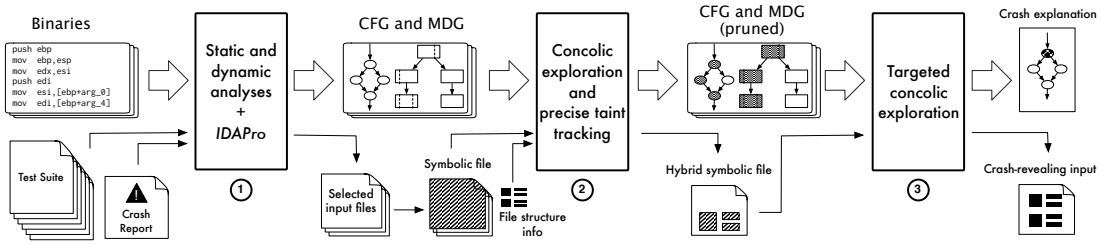


Figure 4-1: An overview of our approach and *Hercules* tool

of programs in a stripped binary is incomplete when collected statically, while recovering such information dynamically is often infeasible.

Reproducing crashes in multi-module systems requires targeted exploration. The search space of potential crashing paths is too large to be exhaustively checked path by path (for instance, using symbolic execution); the complexity of program inputs is too high for exhaustive set of inputs to be generated combinatorially or randomly (for instance, using fuzzing). The space of program paths is intractable for modern analysis techniques – a novel targeted exploration is needed.

Given a real-world program binary with a crash report, our approach *Hercules* tackles the problem of finding program paths and corresponding program inputs that cause a crash in a given program location. The **core idea** behind our approach is to systematically detect, bound and explore a subset of program paths necessary and sufficient for reaching and triggering a given program crash.

The approach builds upon concolic exploration and propagates a necessary, but minimal subset of input data in symbolic form, while keeping the remaining input data concrete. The exploration uses targeted search strategy that helps to explore as few as possible paths, while resolving enough information about program structure for finding the crashing path.

Our approach works in **three main steps** (Figure 4-1). Each step progressively more precisely establishes program and input structures

that are relevant to reproducing the crash. The approach starts with a preprocessing step for initial reconstruction of a program structure and selection of input files (*Step 1*) followed by two passes of concolic exploration. Application of two passes of concolic exploration is a distinct feature of the approach. Each pass serves different purpose: the first pass (*Step 2*) establishes a relationship between program input and relevant program structures, and provides an input to the second pass (*Step 3)* – a focussed fine-granularity search for a crashing path. Our search strategy infers the reasons for infeasibility of specific non-crashing paths, thus helping us to avoid exploring large numbers of paths that are non-crashing for the same reason, and to direct the search towards the paths that will crash the system.

We call our tool and method as *Hercules*, largely because of the Herculean task (of finding crashing inputs) it accomplishes in a reasonable time-frame. This is because of smart search heuristics and structuring of the search into phases. Our approach builds upon selective symbolic execution technique *S2E* [35], extends it, and makes a number of technical contributions, namely -

- *Targeted search strategy* implements our targeted search algorithm that detects reasons for infeasibility of non-crashing paths and directs concolic execution.

- *Approximation of string functions* scales concolic execution by bounding exploration of string manipulation functions that generally cause path explosion.

- *Analysis of loop-controlled crash instructions* enables automatic synthesis of loop-dependent crash conditions.

- *Dynamic module selection* adds flexibility to the *S2E* technique –

in the process of selective concolic execution our technique allows dynamic selection of program structures for concolic execution (state forking).

- *Dynamic CFG refinement.* In addition to the standard *S2E* functionality our technique builds and dynamically refines program control flow graph (CFG) and uses it for reachability analysis to inform concolic exploration.

Assumptions The few significant assumptions we make concern availability of a test suite (a set of non-crashing benign input files) `TestSuite` and indication of a crash location `CL` in a crashing module `CrashingModule`, where execution of at least one test case in `TestSuite` reaches `CrashingModule`. We optionally assume availability of a list of modules invoked during crashing execution `ModuleList`, call stack and values of the program registers at the moment of crash that form a crash condition `CC`. If available, the knowledge about input file structure and layout may aid seed file selection and generation of hybrid symbolic inputs. The required information is external to the approach and can be often produced by a separate static/dynamic analysis.

4.2 Motivating Example

We illustrate the pertinent aspects of the approach using data from a vulnerability `CVE-2010-0718` in Windows Media Player – a buffer-overflow that triggers a system crash in a divide-by-zero exception. Figure 4-2 shows fragments of information used by our approach in the search for crashing input. According to the crash report, the list of modules involved in crashing behavior contains *quartz.dll*, *wmp.dll* and a

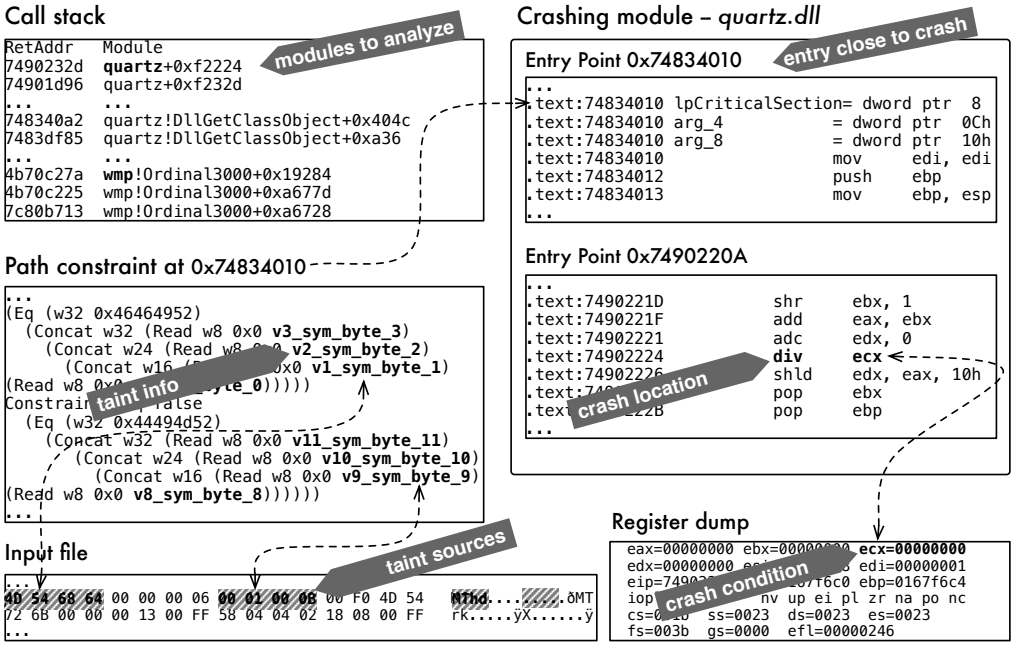


Figure 4-2: Crash analysis information for `CVE-2010-0718`

main module *wmplayer.exe* (out of total 84 modules loaded by the program).[1] The module *quartz.dll* crashes at the program location `0x74902224`, instruction '`div ecx`'. A set of benign inputs does not reach this location.

In *Step 1* of the approach we reconstruct the structure of a system with static analysis and dynamically, by exploring the system with benign input files. The result of this step is an incomplete program structure incorporating module dependence and control flow information. *Step 1* also selects benign inputs that trigger execution in the modules involved in the crash. For `CVE-2010-0718`, *Step 1* identifies a benign input that reaches the crashing module at an entry point (internal function `0x74834010`), but does not reach the crash location. A fragment of the benign file is shown in Figure 4-2.

In *Step 2*, we use concolic exploration as an apparatus for precise taint

[1] We refer to a *module* to denote an executable file (main module) and any library it loads, while for the *entry points* of a module we consider both exported and internal functions.

tracking and identifying input fragments relevant to reaching the crashing module. From these data we generate hybrid symbolic inputs that maintain correct input file structure.[2] For `CVE-2010-0718`, *Step 2* collects a path constraint with a symbolic version of the benign input. The path constraint indicates symbolic bytes from the input file that are propagated to the module entry point `0x74834010` (arrows between the path constraint and the input file in Figure 4-2). These input portions are relevant to reaching crashing module and we mark them symbolic in a hybrid symbolic file.

Generation of hybrid symbolic inputs addresses two main issues in symbolic execution for real-world program binaries. First, hybrid symbolic inputs prompt less constraint solving in concolic exploration and result in smaller symbolic formulae. Second, exploration with structurally correct hybrid inputs has higher chances of bypassing the parser component that incorporates multitude of conditions that cause state explosion in symbolic execution and prevent it from reaching deep program paths.

In *Step 3*, we apply a targeted search strategy (second pass of concolic execution) to explore the system in a directed fashion systematically eliminating groups of paths from analysis and generate crashing input. The strategy stems from the observation that groups of non-crashing paths often have the same *cause* for which they *do not* crash the system. The **main intuition** behind our strategy is that we can detect a reason of infeasibility of a certain path and eliminate from consideration in concolic execution groups of paths that do not crash for the same reason. A contrived example of a shared cause for non-crashing paths is shown in Figure 4-3. Paths through nodes highlighted in yellow (horizontal bars) cannot crash the system because they are guarded by the condition `(x>0)`

[2]We refer to *hybrid symbolic* inputs as files containing fragments of symbolic and concrete data, in contrast with *fully symbolic* files that contain only symbolic data.

that does not satisfy the crashing condition (x<0 ∧ y!=0).

To detect the reason of infeasibility of non-crashing paths, we conjoin the path constraint ϕ for a path that reaches crashing module with the symbolic summary Σ of a crashing module with respect to crash location. Intuitively, terms in path constraint ϕ that contradict terms in symbolic summary Σ are the reasons of infeasibility of a complete path from program entry point to the crash location (term (x>0) in path constraint PC in Figure 4-3).

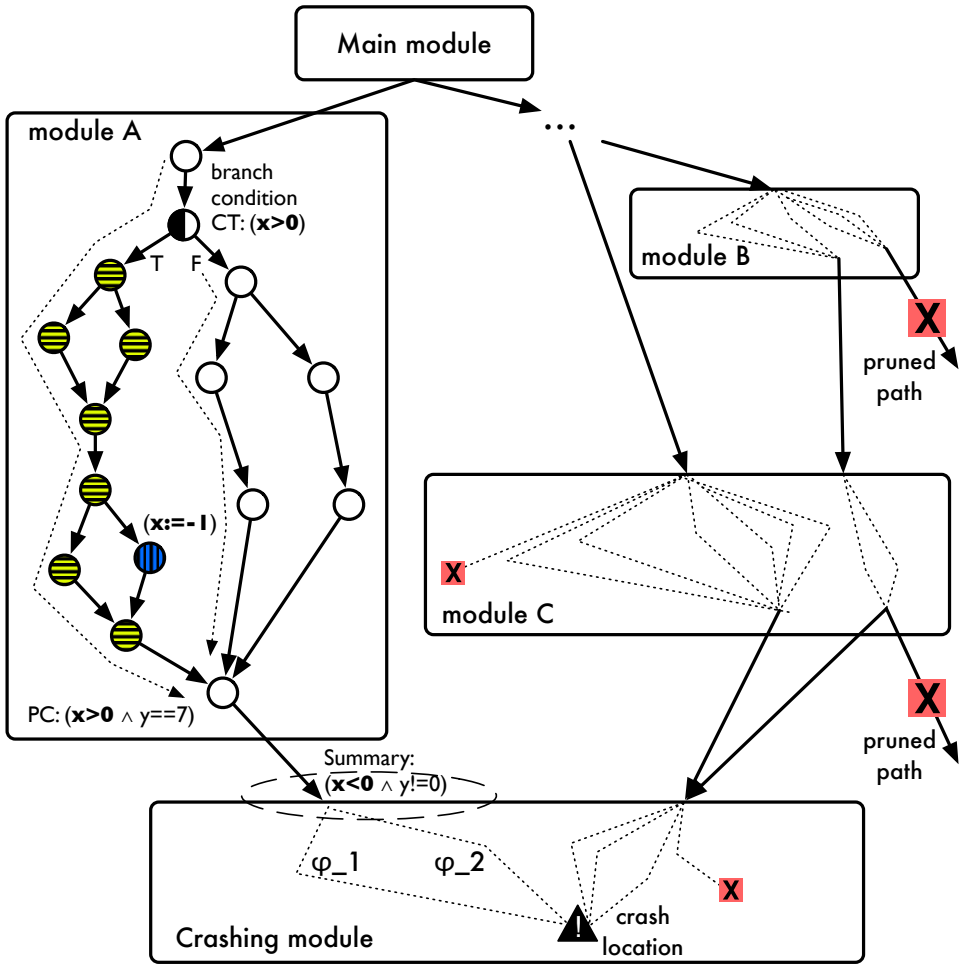


Figure 4-3: Schematic module dependence graph with paths to crashing module highlighted

Practically, the conjoining of ϕ and Σ amounts to two steps. First, to check the satisfiability of formula $\phi \wedge \Sigma$. And second, if the formula is

not satisfiable (the path does not crash the system), to extract a minimal `unsat_core` that contains contradicting terms T. The contradicting terms correspond to the causes of the infeasibility of a given non-crashing path.

Our search algorithm keeps track of each term in path constraint formula and corresponding program location the term is being introduced. Consequently, a contradicting term indicates a point on a program execution path to which our search algorithm proceeds to pursue alternative paths and avoids executing paths that do not crash for the same identified reason.

Crash reports often contain the values of program registers at the moment of crash (register dump in Figure 4-2). A constraint on these values is a crash condition. In case crash condition is available, we can detect the reasons of infeasibility of a path with respect to specific crash condition CC in the same way as described above, by extracting terms in `unsat_core` from an unsatisfiable formula $\phi \wedge \Sigma \wedge CC$.

Depending on a type of a crash, crash conditions are easier or more difficult to extract. For instance, crash due to division by zero could appear in crash report as instruction `‘div ecx‘`, where the value of `ecx=00000000` as shown in Figure 4-2. Consequently crash condition is `(ecx==0)`. Some crash conditions can be less evident and require additional effort for being captured as we discuss in Section 4.5.1.

4.3 Preprocessing and Generating Hybrid Symbolic Files

The first two steps of our approach prepare information for the third step – a targeted search for a crashing path (Section 4.4). In particular, the first step resolves incomplete information about program binaries, while

the second step generates a structurally correct hybrid symbolic input (Figure 4-1).

4.3.1 Recovering Program Structure and Selecting Seed Files

We statically analyze the system with *IDA Pro* toolset[3] (`https://www.hex-rays.com/idapro/`) and use analysis results to obtain a program control flow graph (CFG) and module dependence graph (MDG) that we dynamically refine in the next steps of the approach. The main sources of incompleteness in program binaries are register indirect jumps and calls, and concealed library entry points (non-exported functions). We process the output of *IDA Pro* and statically resolve jump targets for `switch` statements, detect function boundaries and statically imported entry-exit points for the modules in a list of modules involved in a crash.

We execute the system with benign input files to augment the statically collected information with dynamically imported entry and exit points of the modules of the system, concrete targets for branches dependent on indirect register jumps, and resolved register indirect call targets. A resulting aggregated inter-procedural control flow graph connects different modules of the system along the discovered module entry-exit points.

We select seed files from a test suite according to their relevance to the crash and the modules involved in the crashing behavior. The main criteria for file selection are traces of system executions with test files and file structure information. File structure information indicates which objects in the file are required to exercise certain functionality of the system. We

[3] *IDA* is a state-of-the-art multi-processor disassembler and debugger

aggregate the traces of system execution with the preselected test files and obtain a histogram for selecting files that most extensively use modules from `ModuleList`. We use the selected seed files in the next steps of the approach as the most relevant to the crashing behavior.

4.3.2 Generating Hybrid Symbolic Inputs

We use concolic execution to detect fragments of inputs that are relevant to reaching the crashing module – input fragments that propagate data into the crashing module. Taint tracking using concolic execution precisely associates the fragments of program inputs and affected program locations. It is more accurate than "vanilla" taint analysis that traces program paths affected by program inputs, however does not establish which parts of the input are propagated to which program locations.

In concolic execution we apply random exploration strategy – upon branching, the next path to explore is selected randomly, with an exception that paths generated by string functions are selected from groups of paths as detailed in Section 4.5.2. We automatically generate fully symbolic versions of seed files identified in the previous step of the approach and we trace the propagation of symbolic data from these files during concolic execution; execution stops when it reaches `CrashingModule`.

Given a path that reaches crashing module, a path constraint contains symbolic input bytes (taint sources) relevant for reaching this module. Together with the knowledge of input file structure, this information serves to automatically generate hybrid concolic input file that maintains the original file layout.

We prevent random exploration from "drifting" outside modules in `ModuleList` and dynamically refine CFG of the system extending it with information from concolic exploration. Concolic exploration discovers new

paths if it produces new concrete data to take these paths. In particular, if concolic executor reaches register-indirect jump instruction ‘`jmp [eax]`‘ with a *new* concrete value in `eax`, then it may explore a new path spanning from a new jump target. A maximum number of resolved indirect jumps and calls is thus proportional to the number of new paths we can explore with the concolic data. To prepare CFG for the targeted search, we prune it with respect to crash location in crashing module and with respect to exit points that connect modules in `ModuleList`. A schematic module dependence graph with pruned paths is shown in Figure 4-3, where pruned paths are marked with **X**.

4.4 Unsat-core Directed Search Strategy

In the third step of the approach we apply targeted concolic execution to find crashing paths – program paths that crash the system in `CrashingModule`. The targeted exploration works on a pruned version of CFG and hybrid concolic inputs generated in the previous step of our approach. The three phases of the exploration are: (1) *replay*, (2) *summarization*, and a main phase – (3) *targeted search*. Figure 4-4 illustrates transitions between these phases schematically.

The targeted exploration starts by deterministically replaying one of the observed paths to the crashing module with hybrid concolic input (*replay*). Consecutive symbolic exploration symbolically summarizes the crashing module from module entry point to the crashing location using symbolic data propagated to the module from the program input (*summarization*). Finally, a *targeted search* phase selects and traverses alternative program paths in search for crashing paths.

Targeted search phase evaluates the feasibility of a given path with

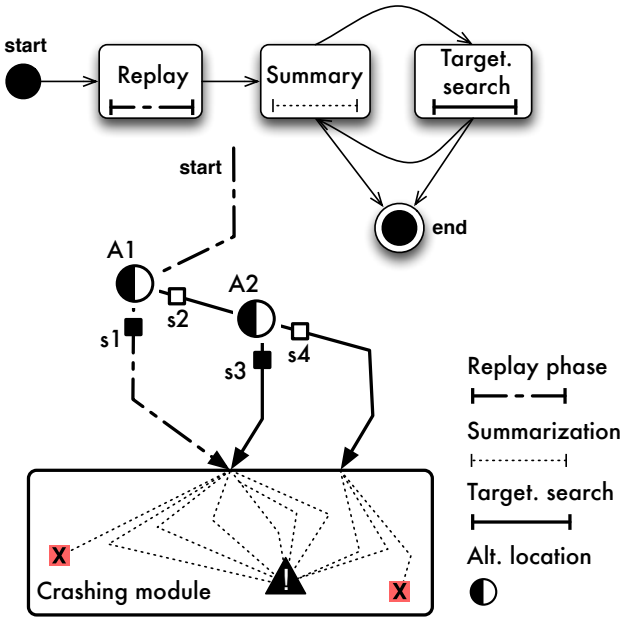


Figure 4-4: Phases of targeted exploration

respect to the crashing module summary and detects the reasons of infeasibility of the non-crashing path as terms in the `unsat_core` of the conjunction of the path constraint and module summary. The algorithm uses the program states that introduced the infeasibility reasons as anchors to select alternative states to which to proceed. As a result, the search is directed away from groups of infeasible paths. Search proceeds until it finds a feasible crashing path or terminates after a user-specified timeout or upon exhausting the memory.

4.4.1 Replay

Algorithm 1 outlines the key elements of each of the phases. The search algorithm is general and can be implemented on top of any dynamic symbolic executor. We illustrate the algorithm for a generic language with instructions identified by their location l. For simplicity we distinguish two types of instructions: (1) branches identified by $branch(l)$ predicate with branch condition $cond(l)$, and (2) non-conditional

instructions. The target location of a branch instruction is identified by $target(l)$, while for all instructions the next location is $next(l)$.

Program state s is represented by a triple (l, ϕ, m), where l is a program location, ϕ is a path constraint, and m is a symbolic store. Symbolic store maps program variables to concrete values or expressions over input variables. The initial program state is $(l_0, true, m)$, where l_0 is a program entry point, path constraint is set to $true$, and m is initialized with symbolic variables for each program input variable.

Targeted exploration replays the path to one of the crashing module entry points $e \in E$. Given a set of states S_e (list of states for reaching e from l_0), the replay is a concolic exploration where upon branching the states for execution are selected from S_e (line 8). During replay the searcher takes snapshots of the alternative states and stores them in a map μ with constraints introduced in the executed state $condition(s)$ (line 9). Figure 4-4 schematically shows state `s1` and its alternative state `s2` that the algorithm stores in the map μ. Replay terminates after traversing all the states in S_e in a state reaching entry point e of the crashing module.

Note that in our implementation of the algorithm, S_e list is lightweight. It does not store the complete state representation as used by *S2E*, but only the forking program locations.

4.4.2 Summarizing Crashing Module Symbolically

Upon reaching entry point of the crashing module, the algorithm commences symbolic summarization (line 11, lines 28–44). The summary is an aggregate of path constraints for each path reaching crash location from the module entry point using symbolic data that is propagated to the module entry point. Symbolic store m holds the propagated symbolic

Algorithm 1 Targeted search

Input: l_0 – initial location; χ – crash location; E – list of module entry points; S_e – list of states for reaching $e \in E$ from l_0;

// REPLAY PHASE:

1: $s \leftarrow (l_0, true, m)$ ▷ Initialize current state
2: **while** $S_e \neq \emptyset$ **do**
3: **if** $\neg branch(l)$ **then** $s \leftarrow (next(l), \phi, m\langle v, e\rangle)$
4: **if** $branch(l)$ **then**
5: **if** $(\text{SAT}(cond(l) \wedge \phi) \wedge \text{SAT}(\neg cond(l) \wedge \phi))$ **then**
6: $s_1 \leftarrow (next(l), cond(l) \wedge \phi, m)$
7: $s_2 \leftarrow (target(l), \neg cond(l) \wedge \phi, m)$
8: $s \leftarrow \{s_1, s_2\} \cap S_e$ ▷ Pick next state from S_e
9: $\mu \leftarrow \mu\langle condition(s), (\{s_1, s_2\} \setminus s)\rangle$ ▷ Snapshot
10: $S_e \leftarrow S_e \setminus s$
11: $\Sigma \leftarrow \text{SymbSummary}(s)$ ▷ Location of the last state in S_e is e

// MAIN PHASE:

12: $X \leftarrow \emptyset$
13: **while** $\neg\text{SAT}(\phi \wedge \Sigma)$ **do**
14: $\tau \leftarrow \text{UNSAT\_CORE}(\phi \wedge \Sigma)$
15: $t \leftarrow pickTerm(\tau, \mu)$ ▷ Pick contradicting term using strategy
16: $X \leftarrow X \cup \{t\}$
17: $s \leftarrow \mu[t]$ ▷ Select alternative state
18: **while** $l \notin E$ **do** ▷ Until reached any of entry points
19: **if** $\neg branch(l)$ **then** $s \leftarrow (next(l), \phi, m\langle v, e\rangle)$
20: **if** $branch(l)$ **then**
21: **if** $(\text{SAT}(cond(l) \wedge \phi) \wedge \text{SAT}(\neg cond(l) \wedge \phi))$ **then**
22: $s_1 \leftarrow (next(l), cond(l) \wedge \phi, m)$
23: $s_2 \leftarrow (target(l), \neg cond(l) \wedge \phi, m)$
24: $s \leftarrow pickNextState(s_1, s_2)$
25: $\mu \leftarrow \mu\langle condition(s), (\{s_1, s_2\} \setminus s)\rangle$ ▷ Snapshot
26: **if** $l \notin E_{checked}$ **then** $\Sigma \leftarrow \text{SymbSummary}(s)$
27: $OUT \leftarrow (\phi, X)$ ▷ We can continue search by proceeding to the remaining alternative states from line 15.

// SUMMARIZATION

28: **procedure** SymbSummary(s) ▷ Explore paths from s to χ
29: Require: $location(s) \in E$
30: $E_{checked} \leftarrow E_{checked} \cup location(s)$
31: $s \leftarrow (l, true, m)$ ▷ Reset path constraint
32: $W \leftarrow \{s\}$ ▷ Initialize worklist
33: **while** $W \neq \emptyset \vee timeout$ **do**
34: **if** $\neg branch(l)$ **then** $W \leftarrow W \cup (next(l), \varphi, m\langle v, e\rangle)$
35: **if** $branch(l)$ **then**
36: **if** $(\text{SAT}(cond(l) \wedge \varphi) \wedge \text{SAT}(\neg cond(l) \wedge \varphi))$ **then**
37: $W \leftarrow W \cup (next(l),\ cond(l) \wedge \varphi,\ m)$
38: $W \leftarrow W \cup (target(l),\ \neg cond(l) \wedge \varphi,\ m)$
39: **if** $l == \chi$ **then** $\Sigma \leftarrow \Sigma \vee \varphi$
40: $W \leftarrow W \setminus s$
41: $s \leftarrow pickNextState(W)$
42: $\Sigma \leftarrow \Sigma \wedge CC$ ▷ Add crash condition to the summary
43: **return** Σ
44: **end procedure**

data as expression over symbolic program inputs. The summary is independent of the path constraint ϕ used for reaching the crashing module and the corresponding path constraint is reset to *true* (line 31). A module summary Σ is a disjunction of path constraints φ_i for each path reaching crashing location χ from a given module entry point: $\bigvee_{i=1}^{n} \varphi_i$.

Summarization procedure uses the pruned CFG to inform selection of the next states in symbolic exploration. States extending outside CFG are not pursued as they do not reach crashing location. This is implemented in procedure $pickNextState()$ that uses CFG to select successors for branching instructions (line 41). This way the algorithm ensures selection of states for paths that reach crashing location.

The symbolic summary collected with our approach may be incomplete. Symbolic data in symbolic execution can be injected only from the input of the system – it is not be generated in the process of symbolic execution. Concolic exploration may not reach the module with symbolic data for all of its inputs, some of the inputs may be reached with concrete data resulting in an incomplete summary.

Symbolic data may not reach the module for a number of reasons. First, seed input files may be inadequate or deficient with respect to the functionality of a crashing module, input file may lack data structures that affect certain input of a crashing module. Second, an input of the module may be independent of the program input. And third, a symbolic input may be concretized during concolic execution and propagated to the module input as a concrete data.

Given a crash condition CC, a module summary Σ is a precondition with respect to reaching crashing location, where $\Sigma(CrashingModule, CC)$ is a logical formula over the module input which is true for all inputs that cause crashing module to reach a final state satisfying CC. Since CFG of

crashing module is pruned, module final state is in the crashing location χ. The algorithm extends module summary Σ with crash condition CC in the last step of summarization (line 42).

The summary Σ concisely captures a precondition for reaching crashing location.

4.4.3 Searching for a Crashing Path

Targeted search phase starts from the point when concolic executor have reached the crashing module in the *replay* phase and consequently collected symbolic summary Σ of the module in the *summarization* phase. *Targeted search* phase identifies program states that *do not* introduce infeasible constraints in the paths reaching crashing module and directs exploration through these states in the search for feasible crashing paths.

Provided that the initial path selected for reaching the crashing module in the replay phase does not crash, the conjunction of path constraint and symbolic summary $\phi \wedge \Sigma$ is unsatisfiable. To detect the reasons of unsatisfiability the algorithm queries SMT solver for minimal `unsat_core` that contains a list of contradicting terms from both path constraint ϕ and summary Σ. The algorithm extracts from `unsat_core` a list of terms τ (line 14). These terms correspond to the reasons for infeasibility that originate from the specific program states on the path reaching crashing module. In the schematic example in Figure 4-3 the cause for the path infeasibility is located by the contradicting term `(x>0)` from the path constraint.

To continue the search for a crashing path, the algorithm selects alternative program states that do not introduce the identified infeasibility reasons. The algorithm selects alternative states indicated by the list of contradicting terms τ using the map of constraints and

alternative state snapshots μ captured during replay phase. In particular, a procedure $pickTerm(\tau, \mu)$ selects one term t from the list τ and this term is then used to query the map μ to select the alternative state (lines 15–17).

In $pickTerm(\tau, \mu)$ we select a term introduced in the top-most program location and a corresponding alternative state. Such term represents a general reason for infeasibility of multiple paths in a symbolic subtree and thus, when selected, can dramatically reduce the search space. However, some of the paths in that subtree may be feasible. For instance, in Figure 4-3, a path that passes through a blue node (vertical bars) in the CFG is feasible with respect to the crashing module summary.

Each iteration of the targeted exploration continues from the selected alternative state until it reaches entry point of the crashing module with a new path constraint ϕ.

The search algorithm can reach crashing module through an entry point that it has not reached before (line 26). In this case the module summary is recomputed to consider new paths to the crashing location, if they are reachable from this entry point.

Algorithm 1 iterates until the formula $\phi \wedge \Sigma$ is satisfiable and hence the crashing path is found. The output OUT of the targeted search consists of a path constraint ϕ and a list of contradicting terms X used for navigating the search. The path constraint ϕ can be solved to generate a set of program inputs that exercise a particular crashing path. The list of selected contradicting terms X serves as an additional explanation for the crashing path highlighting the data and deviation points (`A1` and `A2` in Figure 4-4) that are crucial for pursuing it.

4.5 Tackling Limitations of Concolic Execution

4.5.1 Synthesizing Crash Conditions for Loop-controlled Crashes

To reproduce a crash our approach reaches a crash instruction and, among other information, uses crash condition CC to direct the targeted search and, ultimately, synthesize crashing input. In practice, however, crash condition cannot be formulated symbolically in terms of symbolic input of the program if concolic executor reaches crashing instruction without symbolic data in the operands of the instruction. Previous research demonstrated that this situation can be alleviated for loop-dependent variables [94].

Hercules solves this problem by inferring a function over dependent variables (operands of crash instruction) on a number of loop iterations. This allows us to express the CC at the targeted crash instruction through another condition CC' at the beginning of the controlling loop(s). Saxena et al. used abstract interpretation and pattern matching to infer the function [94]. In *Hercules*, we infer the function using data fitting on runtime values in registers and memory locations during loop exploration [44]. A similar idea has been successfully applied in the context of segmented symbolic analysis to discover symbolic relationships between program variables [69].

Figure 4-5 shows an example of loop-controlled crash instruction in a crash module *flvff.dll* that causes a memory access violation in *Real Player* due to an integer overflow vulnerability (`CVE-2010-3000`). In this example, the crash function is iteratively called in a loop and the crash instruction

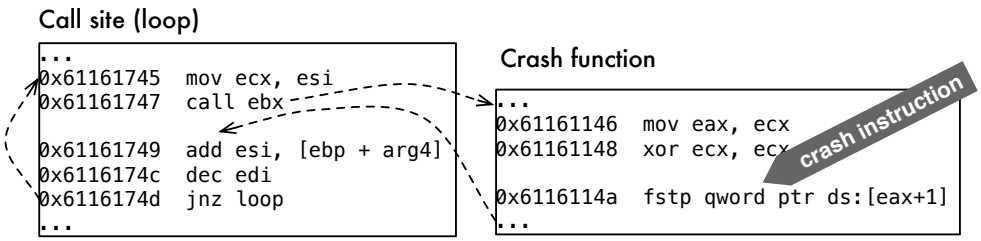


Figure 4-5: Example of loop-dependent crash in Real Player

at `0x6116114a` attempts to store data to the targeted memory address that is calculated using the value of `eax` register. If the address is out of bounds, the crash will occur. Hence, the CC in this example must be expressed through symbolic data in `eax` as $eax == eax_{crash}$, where the eax_{crash} value comes from the register dump in the crash report. However, symbolic execution reaches the crash instruction with a concrete value in `eax` preventing the approach from formulating a symbolic crash condition.

We apply inter-procedural data flow analysis to establish whether the crash instruction is loop-controlled and, if so, to detect data dependencies between the operands of the instruction and the variables within the loop. In the example, we discover a data dependency between `eax` and `ecx` in the crash function (‘`mov eax, ecx`‘ at `0x61161146`) and between `ecx` and `esi` in the call site (‘`mov ecx, esi`‘ at `0x61161745`). Inside the loop, `esi` is incremented by a concrete value (passed through a function argument) at each iteration. The value of `eax` in crash instruction depends on the value of `esi` register inside the loop and in turn, the value of `esi` depends on the number of loop iterations.

Using data fitting, *Hercules* infers a relationship between `esi` and a loop count `it`: $\texttt{esi} = \texttt{esi}_0 + \texttt{it}*0x23$. In the example, the number of loop iterations is controlled by the value of register `edi` that holds symbolic data (instructions at `0x6116174c` and `0x6116174d`). In other words, the value of `eax` in crash instruction is indirectly controlled by the symbolic

input data in `edi`.

As a result, we transform the concrete constraint CC on `eax` at the crash instruction to a symbolic constraint CC' on the value of `edi` before the start of the loop. With this data we can synthesize crashing input by solving the formula $\phi' \wedge CC'$, where ϕ' is the path constraints to reach the loop.

4.5.2 Path Grouping in String Manipulation Functions

To avoid path explosion during concolic execution of real-world binaries our approach tackles its most prevalent sources – loops and string manipulation functions. To tackle path explosion in loops we bound a number of loop iterations in which concolic executor forks new feasible states. Beyond the bound the executor does not fork new states in a loop. Recent research shows that bounding loop iterations is a practical and effective solution in the context of symbolic execution [105].

String manipulation functions are more difficult to tackle than loops. In essence these functions are sophisticated loops over string data that are modelled with bit-vectors and processed as unbounded data causing generation of infinitely many symbolic states. Yet, symbolic exploration with string data is important, a large class of crashes in software is caused by buffer- and heap-overflows when programs operate on string data.

We define a heuristic that leverages string length estimation and approximation of standard string manipulation functions to help concolic execution in generating states with realistic string data while reducing the risk of path explosion. The intuition behind our heuristic comes from the following observations: there are many concrete strings encoded in the

program code and thus many string length bounds can be obtained based on operations between symbolic and concrete strings. Moreover, there are practical limitations on the size of strings such as function stack frame size and input file layout that provide estimates of string lengths. Finally, semantics of several standard string manipulation functions can be abstracted to the level of groups of paths and inform symbolic execution.

For functions like `strlen(sym)` we bound concolic exploration in the function according to the length estimate of a symbolic string parameter `sym` that we gather dynamically from a number of sources. A length estimate for strings allocated on stack should not exceed a current stack frame size, while file layout and object boundaries (boundaries between symbolic and concrete input data) indicate upper bounds for lengths of strings derived from input file data.

For other standard string functions that operate on pairs of strings we approximate these functions by mapping their few semantically different high-level paths to a multitude of low-level paths. One example of groups of high-level paths for a function `stricmp(str1,str2)` would be: (1) strings are equal, (2) strings are equal length and differ in content, and (3) strings differ in both length and content. These three groups map to thousands of feasible low-level paths stemming from two reasons. First, in LLVM based symbolic execution engine – *S2E* in our case – the string function is converted to LLVM bitcode that has larger number of branch instructions that in source code or binary. For `stricmp(str1,str2)` function the number of branches in LLVM bitcode is 13 versus 3 in source code. Second, the number of paths is also controlled by the number of loop iterations that depends on the length of the input strings. We define the high-level semantics of the string functions as a logical formula over function input, output and properties of the input, such as length of a

string argument.

To avoid path explosion, during concolic execution we bound the exploration of these functions until paths from all semantically different path groups are generated, while controlling the number of generated paths. Consecutively, we prioritize groups of paths and select single paths from each group for further concolic exploration. For instance, for `stricmp()` function we give a higher priority to the path producing equal strings which covers the highly relevant case.

An experimentation with Orbital Viewer case study (`CVE-2010-0688`) highlights the degree of reduction in path numbers our technique achieves for concolically exploring a standard string function. *S2E* with depth-first search configuration would need to fork $(2^{13}) * 18 \approx 150$K paths to fully concolically explore `stricmp(str1,str2)` function with one symbolic string argument and one concrete string of length 18.

With our heuristic concolic executor only needs to explore 8K paths to populate elements for three high-level groups of total 19 paths that we keep: one path (strings are equal), one path (strings of equal length and differ in content), and 17 paths (strings differ in both length and content). Each path in the third group corresponds to the strings being unequal in any of the first 17 characters. We only need to keep 19 paths to cover all of the three high level paths of the `stricmp(str1,str2)` function, while the remaining low-level paths can be removed from exploration. Overall, we generate few paths that cover all high-level paths of a function in a balanced way and produce realistic strings.

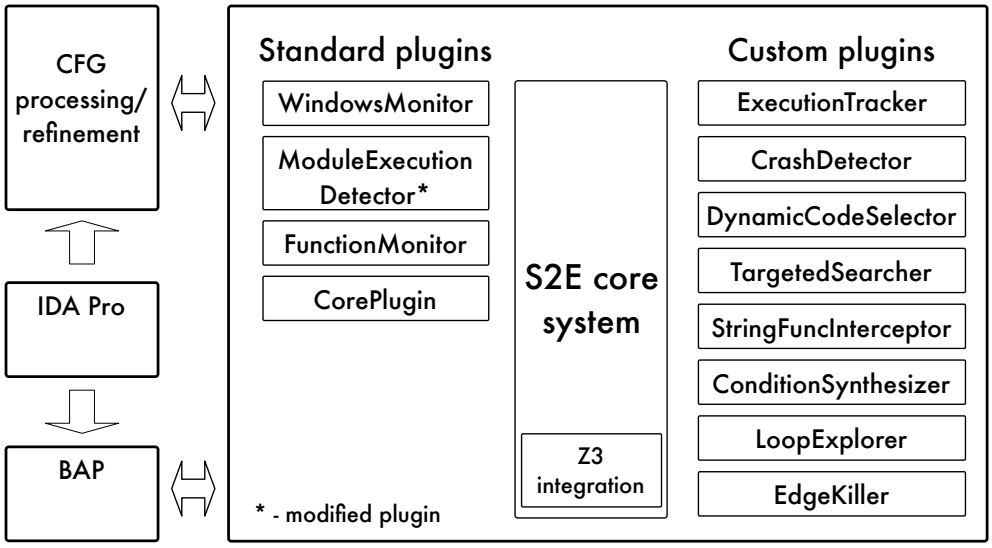


Figure 4-6: Components of the *Hercules* toolset

4.6 Implementation

Our approach *Hercules* builds upon and extends the selective symbolic execution technique *S2E* [35]. Figure 4-6 shows an overall view of the components of our toolset. The main components of our system are built as custom *S2E* plugins. In addition, *Hercules* provides tools for control flow graph processing outside *S2E* and data flow analysis built on *BAP*.

4.6.1 CFG Refinement and Path Pruning Functionality

Hercules implements analyses for post-processing the output of *IDA Pro* toolset and obtaining the static and the dynamic program structure information. We build CFG for each selected module of the system using the static program structure information (direct jumps, direct calls, jump tables) and the dynamic information (indirect register jumps and calls).

We refine the CFG whenever the dynamic program structure is updated, while exploring the program under test in *Steps 2* and *3* of our approach.

A *PathPruner* module implements a pruning algorithm similar to the

algorithm for computing "chop" by Brumley et al. [23]. *PathPruner* indicates every path that *does not* lead to interesting targets in a module dependency chain. In the crashing module, this tool will prune the paths that do not reach the crash location. The output of the tool will be used as the input of a plugin *EdgeKiller* that will kill a *S2E* state in runtime if it executes an undesirable path.

4.6.2 Extensions of the S2E Core

STP, the SMT solver in *S2E*, does not compute `unsat_cores`. To get the `unsat_core` of a symbolic expression we integrate *Z3* with *S2E* and pass symbolic constraints between them in SMT2 format. Our framework augments *S2E* to output symbolic formulae in SMT2 format and implements a wrapper function to invoke *Z3* solver from *S2E*.

Another *S2E* core update takes snapshots of *S2E* states in the targeted search. We make snapshots of symbolic states at each branch location during concolic execution to enable backtracking of concolic executor. This functionality is implemented on top of cloning functionality of *KLEE* used by *S2E* and our version supports state cloning at an arbitrary execution point.

4.6.3 Analysis and Search Plugins

An *ExecutionTracker* plugin outputs important runtime information. It handles signals emitted by *S2E core* plugin when it executes an instruction or a basic block. In addition, it detects the Process_ID of the program under analysis to keep track of the information it produces and excludes information produced by other programs that use shared libraries.

A *DynamicCodeSelector* plugin enables flexible runtime selection of

modules executed concolically (with forking enabled). The original *S2E* CodeSelector plugin is less flexible and only supports static configuration of a list of modules in which *S2E* selectively enables forking. Our *TargetedSearcher* plugin heavily relies on *DynamicCodeSelector* for dynamically switching different search stages each having different configurations of forking-enabled modules.

To synthesize crash conditions for loop-controlled crash instructions, we have developed three components. First, *ConditionSynthesizer* is built as *S2E* plugin. It outputs runtime values of all registers and updated variables at each iteration inside the controlling loop. Second, a light-weight data flow analysis is built on Binary Analysis Platform (*BAP*) [24]. Its output supports user in selecting registers/variables having relationship with a number of loop iterations. Third, a tool to interface with the *R* statistical package to invoke its regression models and infer function on dependent registers/variables and the number of loop iterations [90]. *Hercules* infers functions for simple and nested loops and covers three function forms – linear, polynomial, and exponential – by using simple linear and multiple linear regression models with logarithm and variable substitution transformations.

StringFunctionInterceptor controls the exploration inside string functions. It intercepts every call to the list of standard string library functions such as `strlen`, `strcpy`, `strcmp`, `stricmp`, `strcat`, `strchr` and `strstr` using handling signals emitted by the *FunctionMonitor* plugin of *S2E* (`onFunctionCall` and `onFunctionRet` signals). For each of the functions we implement a special structure to define groups of semantically distinct high-level paths (Section 4.5.2). Each group is defined as a logical expression over function input, lengths of manipulated strings and function output. Finally, the module dynamically extracts the

stack frame size of the caller to estimate string length bounds.

A *TargetedSearcher* plugin is a combination of the three searchers (1) *PathReplaySearcher*, (2) *SymbolicSummarization* and (3) *EntryPointTargetedSearcher*. Each searcher implements the dedicated phases of the targeted search algorithm defined in Section 4.4. The plugin switches between the searchers in the process of concolic execution using several signals emitted by the *S2E Core* plugin (`onStateFork`, `onStateSwitch`, `onExecuteInstruction`) and signals from our custom plugins. In particular, `onStringFunctionStart` and `onStringFunctionEnd` signals generated by the *StringFunctionInterceptor* plugin are used for state grouping and prioritization for string manipulation functions. *TargetedSearcher* populates the groups of states defined for each string function, prioritizes these states and removes redundant ones.

A *CrashDetector* module detects application crash by tracking Windows error reporting service invocation and calls *S2E* API to solve path constraint and generate crashing input.

4.7 Experimental Evaluation

We evaluated our approach experimentally on real-world application binaries. In this section we present the results of the evaluation that demonstrate that *Hercules* successfully reproduced *six* distinct crashes in five applications: Adobe Reader (AR), Windows Media Player (WMP), Real Player (RP), Orbital Viewer (OV) and Music Animation Machine (MAM) Player. Table 4.1 summarizes the results for the effectiveness of our approach as compared to the original *S2E* technique and widely used industrial black-box fuzzing tool *PeachFuzzer*

Table 4.1: Experimental setup and results

CVE IDs	2014-2671	2010-0718	2010-0688	2011-0502	2010-2204	2010-3000
Application	WMP v9.0	WMP v9.0	OV v1.04	MAM v0.35	AR v9.2	RP SP 1.0
Selected / total modules	4 / 84	3 / 86	2 / 49	1 / 51	2/78	2/129
Size of crash. module	1.22 MB	1.22 MB	538 KB	368 KB	2.32 MB	60 KB
Test suite – No. of files	10	15	10	10	5	6
Test suite – file size	2–137K	2–54K	3–5K	2–5K	55–307K	87–654K
S2E (Random search)	✘ (>12 hr)	✘ (>12 hr)	✘ (>12 hr)	✔ (2 min)	✘ (>12 hr)	✘ (>12 hr)
S2E (DFS search)	✘ (>12 hr)	✘ (>12 hr)	✘ (mem exh.)	✘ (>12 hr)	✘ (>12 hr)	✘ (>12 hr)
PeachFuzzer	✘ (>24 hr)	✘ (>24 hr)	✔ (10 hr)	✔ (10 min)	✘ (>24 hr)	✘ (>24 hr)
Hercules ①	5 min	5 min	2 min	1 min	5 min	5 min
Hercules ②	45 min	90 min	120 min	∼0 sec	120 min	120 min
Hercules ③	✔ (15 min)	✔ (60 min)	✔ (40 min)	✔ (30 sec)*	✔ (60 min)*	✔ (45 min)

(`http://peachfuzzer.com`). *Hercules* generated test inputs and reproduced all six crashes, whereas baseline techniques failed or took considerably more time to succeed.

4.7.1 Experimental Setup

We conducted all of the experiments on a computer with a 3.4 GHz Intel Core i7-2600 CPU and 8 GB of RAM. The host OS is Ubuntu 12.04 64-bit. The guest OS are *Windows 7* Enterprise 32-bit SP1 and *Windows XP* 32-bit SP3. Our approach is implemented on *S2E* version from May 2, 2014 obtained at `https://github.com/dslab-epfl/s2e`. We used freeware *IDA Pro* 5.0 to disassemble binaries. In Table 4.1, case studies marked with (*) have been tested on both Windows XP and Windows 7.

The case studies cover vulnerabilities of the four prevalent types (buffer overflow, integer overflow, memory access violation and division-by-zero) from `http://cve.mitre.org/` and operate on five distinct structured file formats. For the OV case study we used a developer test suite obtained at `http://www.orbitals.com/orb/ov.htm`. For the Adobe Reader case

study, we used Microsoft Word 2010 to create pdf files with embedded fonts. For the other four case studies, we obtained test suites on the Internet from a random sample of benign files of an appropriate format. Table 4.1 highlights the test suite composition with numbers of benign files and their variation in size. We enabled forking in a subset of modules indicated by the crash reports as shown in Table 4.1.

The toolset was configured for a timeout after twelve hours of exploration and run without parallelization of the execution process. For *Hercules*, we have fixed a loop bound of three iterations and state timeout of 30 seconds to prevent the exploration from "drifting" (Section 4.3).

Table 4.1 shows execution times for the CFG construction (*Step 1*), concolic (*Step 2*) and the targeted (*Step 3*) exploration by *Hercules* as per Figure 4-1 (correspondingly marked ①,②,③ in the table). For *Step 1*, our automated scripts construct CFG from the output of *IDA Pro* within few minutes. We used a practical time limit of two hours for exploration in *Step 2*. For the five case studies (except `CVE-2011-0502`), *Hercules* (*Step 2*) explored, resolved dynamic information and reached a crashing module within two hours, while the case study on MAM (`CVE-2011-0502`) did not require exploration phase to reach the crashing module. Finally, for all the case studies targeted search (*Step 3*) reproduced the crashes within an hour.

4.7.2 Reproducing Crashes

Our approach reached and reproduced crashes `CVE-2014-2671` and `CVE-2010-0718` in Windows Media Player (Quartz library). `CVE-2014-2671` is a vulnerability in Windows Media Player version 9. Attackers can exploit this vulnerability to cause a denial of service via a crafted `.wav` file. `CVE-2010-0718` is a buffer overflow vulnerability in

Windows Media Player version 9. It allows attackers to cause a denial of service via a crafted `.mpg` or `.mid` file that triggers a system crash due to a divide-by-zero exception. *Hercules* successfully reproduced the two crashes using the targeted search. In both cases, *Hercules* avoided state explosion by bounding loop iterations, while no string function analysis was required. *Hercules* reproduced `CVE-2010-0718` using as little as 1% of input data in symbolic form.

`CVE-2010-0688` is a crucial stack-based overflow in Orbital Viewer, a tool for visualization of atomic and molecular orbitals. By using a crafted `.orb` or `.ov` file, attackers can trigger a system crash in Memory Access Violation exception or execute arbitrary code. The vulnerability comes from the code for reading data from input file using a known vulnerable function `fscanf`. OV does not correctly check the data size before writing it into stack buffers. The crash happens when the overwritten data section is accessed by OV after a series of function calls, including calls to string manipulation functions. *Hercules* successfully bridged the distance between the location where crashing data is introduced and the crashing location by leveraging our heuristic for exploring string functions (Section 4.5.2), and reproduced the crash.

`CVE-2011-0502` is a vulnerability in MAM MIDI Player that allows attackers to easily cause a denial of service via a crafted `.mid` file that crashes the program with a null pointer dereference. This is the most "simple" case study in our experiments that *Hercules* reproduced within 30 seconds. Furthermore, *Hercules* does not require loop bounding nor string function analysis to reproduce the crash.

`CVE-2010-2204` is an vulnerability in Adobe Reader 9.0–9.3 that allows attackers to cause a denial of service or execute arbitrary code. `CVE-2010-3000` is an integer overflow vulnerability in RealPlayer SP 1.0

that allows attackers to execute arbitrary code. For both cases *Hercules* can reach crash instructions by symbolically executing the programs with benign inputs, however, the programs do not crash, because the crash instructions are loop-controlled. With the loop-controlled crash condition analysis (Section 4.5.1) *Hercules* can identify the loop and infer a relationship between crash instructions and the controlling loops. As a result, *Hercules* can successfully synthesize symbolic crash conditions on the number of loop iterations and use them to reproduce both crashes.

4.7.3 Comparing with the Baseline

We demonstrate the effectiveness of *Hercules* by comparing it with the baseline *S2E* and black-box fuzzing tool *PeachFuzzer* on the same six case studies. We have run *S2E* with the input files that *Hercules* used to successfully reproduce the crashes, while *PeachFuzzer* used all the files in each test suite.

The results shown in Table 4.1 demonstrate that *S2E* can reproduce the "simple" crash (`CVE-2011-0502`) and fails to reproduce the other ones. Non-directed search of the baseline *S2E* prevents it from reaching relevant program locations in a given time and state space constraints. When run with a depth first search (DFS) exploration, *S2E* digs itself in a single path, while for the OV case study (`CVE-2010-0688`) it gets path explosion in string manipulation functions before reaching the crash location.

We run *PeachFuzzer* in a fully automatic setting with infinite iterations of random mutation strategy and without user-provided data model (input grammar specification) for up to 24 hours. *PeachFuzzer* took substantially more time than *Hercules* to generate crashing inputs for two case studies. Effectiveness of the fuzzing tool critically depends on the results of manual analysis to provide it with a correct input grammar

specification and indicate input portions that can and must be mutated, and portions that need to be preserved.

4.8 Chapter Summary

In this chapter, we have presented the design and evaluation of our *Hercules* approach for finding test inputs which can reproduce a given crash. Our approach is based on symbolic execution and its distinctive features include (i) working on binaries without source code and encompassing techniques to construct the control-flow graph directly from binaries in the presence of register-indirect jump instructions, (ii) combining taint tracking and symbolic execution to find which parts of the input file must be kept symbolic, and (iii) search strategies to direct a path towards the crashing location by analyzing why the current path being traversed by the search cannot reach the crash. Experiments on real-world application binaries such as Windows Media Player and Adobe Reader, show the efficacy of our approach in finding test inputs to reproduce a crash.

Chapter 5

Closed-loop Model-based Black-box and White-box Fuzzing for Program Binaries

Many real-world programs take highly structured and complex files as inputs. The automated testing of such programs is non-trivial. If the test does not adhere to a specific file format, the program returns a parser error. For symbolic execution-based whitebox fuzzing the corresponding error handling code becomes a significant time sink. Too much time is spent in the parser exploring too many paths leading to trivial parser errors. Naturally, the time is better spent exploring the functional part of the program where failure with valid input exposes deep and real bugs in the program.

In this chapter, we suggest to leverage information about the file format and data chunks of existing, valid files to swiftly carry the exploration beyond the parser code. We call our approach Model-based Blackbox and Whitebox Fuzzing (MoBWF) because the file format input model of blackbox fuzzers can be exploited as a constraint on the vast

input space to rule out most invalid inputs during path exploration in symbolic execution.

5.1 Introduction

Testing file-processing programs can be challenging. Even though a structured file is stored as a vector of input bytes, it is often parsed as a tree where data chunks contain fields and other data chunks.

Our key insight is that certain branches in a file-processing program are exercised only depending on i) the *presence* of a specific data chunk, ii) a *specific value* of a data field in a data chunk, or iii) the *integrity* of the data chunks. Hence, an efficient test generation technique not only sets specific values of the fields but also adds/removes complete chunks and establishes their integrity (e.g., checksum or size).

Fuzzers help to test such file-processing programs. Model-based blackbox fuzzers (MoBF) [9, 11] utilize input models to generate *valid* random files. The input model specifies the format of the data chunks and integrity constraints. However, while valid, the modification is still inherently random. Whitebox fuzzers (WF) employ symbolic execution to explore program paths more systematically. Given a valid file, they can generate the specific values for the data *fields* quite comfortably. However, when it comes to adding or deleting data *chunks* or enforcing integrity constraints, they are bogged down by the large search space of invalid inputs [102].

Grammar-based whitebox fuzzers (GWF) can generate files that are valid w.r.t. a context-free grammar [49]. Like WF, GWF computes path constraints: logical formulas that are satisfied only by new files exercising *alternative* paths. Unlike WF, these constraints are converted into regular

expressions such that a context-free constraint solver can generate an input that is accepted by both, the grammar and the expression. However, the expression is much weaker than the path constraint. Suppose, symbolic execution yields the path constraint $\varphi \wedge (x < y)$. After conversion, the regular expression cannot capture that arithmetic constraint. Moreover, GWF cannot encode integrity constraints such as size-of, offset-of, length-of and checksums. These integrity checks are very common in several highly structured file formats like PNG, PDF and WAV.

In this work, we present *Closed-loop Model-based Blackbox and Whitebox Fuzzing* (MoBWF), an automated testing technique for industrial-size program binaries that process structured inputs. MoBWF is a marriage of model-based blackbox fuzzing and whitebox fuzzing that generates valid files efficiently and exercises critical target locations effectively. It is a *directed path exploration* technique that prunes from the search space those paths that are exercised by invalid, malformed inputs: (i) MoBWF uses information about the file format to explore those branches that are exercised depending on the presence of specific chunks. To this end, MoBWF removes the referenced chunk or adds a new valid chunk by instantiation from the input model or a process we call *data chunk transplantation* — MoBWF identifies the set of input bytes corresponding to the required chunk in a donor file and transplants them into the appropriate location of the receiving file. (ii) MoBWF employs selective symbolic execution [35] to explore those branches that are exercised depending on specific values of the data fields. (iii) Lastly, MoBWF establishes the integrity of the generated files, repairing checksums and offsets.

Unlike MoBF, MoBWF is directed and enumerates the specific values of data fields more systematically. Unlike WF, MoBWF does not get bogged

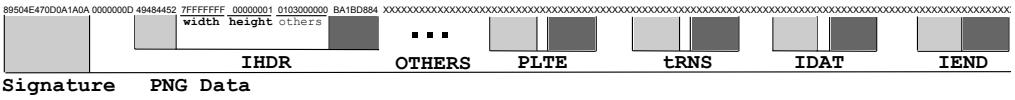


Figure 5-1: The structure and the hex code of a PNG file. A data chunk is a section in the hex code embedding one piece of information about the image. The hex code above the light-grey boxes identifies the data chunk type while the hex code above the dark-grey boxes protects the correctness of the data chunk (via checksum).

down by the large search space of invalid inputs or require any seed inputs (cf. [46, 53]). Unlike GWF, MoBWF maintains *full* path constraints so it has no impact on the soundness and completeness of WF. Moreover, MoBWF leverages a more expressive yet simple input model to handle integrity constraints.

The input model is used to generate valid files efficiently, enforce integrity constraints, and facilitate the transplantation of data chunks. Since it only prunes search space, the input model does not need to be complete. On one hand, whitebox fuzzing eventually constructs all relevant (semi-) valid files by exploring paths that are not pruned by the input model. On the other hand, transplanting data chunks from donors maintains underspecified integrity constraints, such as the concrete compression algorithm with which the image data in a PNG file must be encoded. An input model is constructed once and can be used across all future testing sessions. It has been shown that input models can also be derived in an automated fashion [70, 61, 60]. Each of our input models was constructed manually in less than a day.

The two *main challenges* of Traditional Whitebox Fuzzing (TWF) that we address are:

- **Path Explosion**. Parser code is often a complex part of a program. In practice, TWF gets bogged down by an exponential number of paths in the parser that are exercised by invalid inputs [102].

- **Seed Dependendence**. Most TWF approaches assume the existence of a seed file that features all necessary data chunks – it is only a matter of setting the correct values for the data fields to expose an error. In practice, however, this may not be the case. Data chunks may be missing or in the wrong order. In other cases no seed files may be available at all.

The main contributions of MoBWF are as follows.

- **Pruning Invalid Paths**. The input model allows to prune most paths that are exercised by invalid inputs. As opposed to TWF, MoBWF is capable of negating those *crucial* branches that are exercised only in the presence of certain data chunks without having to iteratively construct the data chunk by exploring the parser code. All generated test inputs are valid in that they adhere to the input model. Integrity constraints are enforced. Given a 24h time budget, our MoBWF tool exposed all of thirteen vulnerabilities in our experimental subjects while the TWF tool exposed only six.
- **Reduced Seed Dependence**. The instantiation from the input model allows to construct seed inputs from scratch. Moreover, given a seed input that is missing a data chunk to reach a target location, MoBWF allows to utilize other seed files as donors, transplant the missing data chunk, and construct a new seed input that is closer to the target location. In the absence of a donor, the missing data chunk can be directly instantiated from the input model. Out of the thirteen vulnerabilities in our experimental subjects our MoBWF tool exposed nine *without any seed inputs*.
- **Fuzzing tool**. We implement our MoBWF tool as an extension of the TWF tool, Hercules [88]. We compare our MoBWF tool not only to the Hercules TWF but also to the Peach model-based

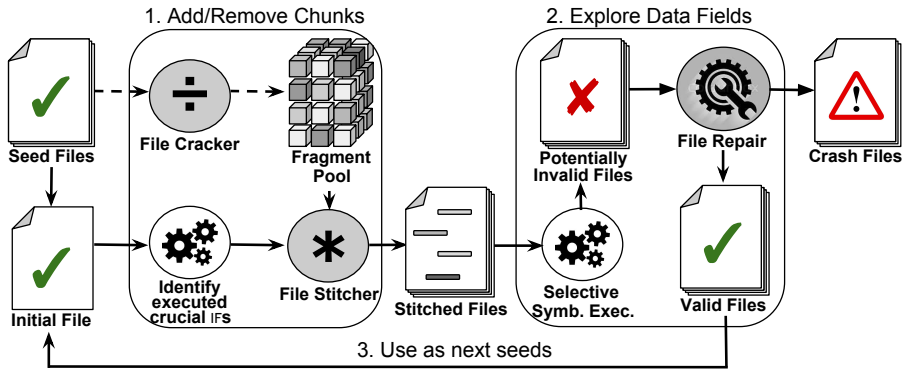


Figure 5-2: Closed-loop Model-based Blackbox and Whitebox Fuzzing. Elements marked in grey are informed by the data model.

blackbox fuzzer [9]. Given a 24h time budget, our MoBWF tool exposed all of 13 vulnerabilities in our experimental subjects while the both HERCULES and PEACH tool exposed only six.

Insights. Through our experiments we also gain insights about the relative strengths of our technique MoBWF, symbolic execution based traditional whitebox fuzzing (TWF), and model-based blackbox fuzzing (MoBF) as in fuzzers like Peach/Spike [9, 11]. TWF performs well only if there exists a seed input that features all necessary data chunks and only certain values for data fields need to be set. MoBF performs well if the vulnerability is exposed by putting boundary values for certain data fields, or by removing/adding empty data chunks. Deep vulnerabilities that require specific values are best exposed by a symbolic execution-based approach. MoBWF performs well even in the absence of seed inputs and swiftly generates the specific values needed to expose even deep vulnerabilities, while also gaining the capability to add and remove complete data chunks as in MoBF.

5.2 Motivating Example

We motivate MoBWF based on a real, serious vulnerability in a library that is shipped with several browsers and media players. LibPNG [97] is the official PNG reference library; it supports almost all PNG features, and has been extensively tested for over 20 years. The library is integrated into popular programs such as VLC media player, Google Chrome web brower and Apple TV.

PNGs consist of four mandatory and fourteen optional types of data chunks. For easy parsing and error detection the file format requires to specify the size, type, and checksum of each data chunk besides the actual data. The particular PNG file in Figure 5-1 happens to expose a memory access violation vulnerability (OSVDB-95632) in VLC 2.0.7 [99] which uses LibPNG 1.5.14. To trigger the bug, the image width defined in the IHDR chunk must take a specific value (from `0x7FFFFFF2` to `0x7FFFFFFF`) and the optional tRNS chunk must exist. The tRNS chunk specifies alpha values to control the transparency of pixels in the image.

Figure 5-1 partially shows structure of a file that exposes the bug. The first eight bytes identify the file as PNG. The next four bytes specify the *size* of the next data chunk (`0xD` = hex(13) bytes), followed by four bytes identifying the *type* of the chunk as IHDR (light-grey box). The next 13 bytes are data fields specifying image width and height. This is followed by four bytes of *checksum* protecting the correctness of the IHDR chunk (dark-grey box). The remaining chunks are structured similarly. The image data in the IDAT chunk is compressed using the DEFLATE compression algorithm [3] and the end of the PNG file is indicated by IEND chunk.

Listing 5.1 shows the pertinent code in LibPNG. In each iteration, `png_read_info` (lines 2-27) parses information about the current chunk,

like its size and type. Depending on the type it calls the corresponding function to handle the current chunk and validate the checksum.

```
1  // read chunks' info before first IDAT chunk
2  void png_read_info(png_structp ptr)
3  {
4  // read and check the PNG file signature
5  read_sig(f);
6  for (;;)
7  {
8    // get current chunk's information
9    uint_32 length = read_chunk_header(ptr);
10   uint_32 chunk_name = ptr->chunk_name;
11   // mandatory chunks
12   if (chunk_name == png_IHDR)
13      handle_IHDR(ptr, length);
14   else if (chunk_name == png_IEND)
15      handle_IEND(ptr, length);
16   else if (chunk_name == png_PLTE)
17      handle_PLTE(ptr, length);
18   else if (chunk_name == png_IDAT)
19   {
20      ptr->idat_size = length;
21      break;
22   }
23   // optional chunks
24   else if ...
25   else if (chunk_name == png_tRNS)
26      handle_tRNS(ptr, length);
27   else if ...
28 }
29 }
30
31 // initialize row buffer for reading data from file
32 void png_read_start_row(png_structp ptr)
33 {
34    size_t buf_size;
35    ...
36    buf_size = calculateBufSize(ptr);
37    ptr->row_buf = png_malloc(ptr, buf_size);
38    png_memset(ptr->row_buf, 0, ptr->rowbytes);
39 }
```

Listing 5.1: Simplified parser code for data chunks. The code is shown to ease the explanation; MoBWF works directly with program binaries.

These handler functions parse a chunk's data fields and store their values

for further image transformation and processing steps. The chunks are parsed until the first IDAT chunk is reached (lines 18-22). The file shown in Figure 5-1 passes all checks in the parser and chunk-handling code and is therefore *valid*.

When all other chunks have been parsed, LibPNG starts reading pixel data from IDAT chunks. For each image row, LibPNG allocates and initializes a buffer (lines 31-38 in *png_read_start_row*). This is the faulty function. Specifically, the existence of tRNS chunk and the improper validation of large image width leads to an integer overflow while LibPNG is calculating buffer size for each row (as simplified in *calculateBufSize* at line 35). Because of that the allocated buffer is much smaller than required (line 36). As a consequence, a buffer overflow occurs in *png_memset* causing the program to crash. Notice that the third argument for the function call *memset* (*ptr→rowbytes*) is much larger than the size of the buffer.

5.2.1 Exposing Vulnerabilities

Traditional Whitebox Fuzzing

Given a benign PNG file having the required data chunks in Figure 5-1 and the dangerous location in *png_memset*, a Whitebox Fuzzing (TWF) tool can automatically generate an input that exposes the vulnerability. However, suppose the benign file is missing the tRNS chunk, it will be an obstacle for TWF because it is very unlikely that TWF can correctly synthesize the missing chunk and keep the file valid. In fact, if there is no tRNS chunk, the true branch of the IF-statement in line 25 of Listing 5.1 is not taken. Although TWF can negate the branch and get a chunk with the name "tRNS", its size and content still adheres to specification of another chunk.

Where LibPNG expects the size, data, and checksum of the new tRNS chunk, it only finds "random noise". So, TWF overrides perfectly encoded image data only to spend *substantial time* constructing a valid tRNS chunk in its place. Since IDAT chunk is compulsory, TWF spends even more time navigating the space of invalid inputs to construct another IDAT chunk until it finally constructed a valid file that contains a valid tRNS chunk and all compulsory chunks where all integrity constraints are satisfied.

Model-based Blackbox and Whitebox Fuzzing

We propose MoBWF as a marriage of model-based backbox fuzzing and whitebox fuzzing. The model-based approach allows MoBWF to cover the search space of *valid* test inputs efficiently while the whitebox approach in detail covers each subdomain more effectively. Both approaches are integrated in a feedback loop that is described in Figure 5-2.

Setup. In this example, the user provides the buggy VLC binary, a crash report, a set of existing benign PNG files (if available) and a PNG model as shown in Listing 5.2. To implement MoBWF, we leverage a *model-based* blackbox fuzzer. The Peach framework allows to specify a file format as Peach Pit [10]. It describes the types of and relationships (size, count, offsets) between data chunks and fields. It also supports fixups and transformers. Fixups allow to repair related data fields, such as checksums. Transformers are used for encoding, decoding and compression.

The PNG Peach Pit in Listing 5.2 first specifies the generic data chunk (lines 1-14). PNG chunks all contain at least three data fields, specifying the length, type, and checksum of the data chunk. The other data chunks inherit these attributes (lines 15-31), fix the chunk type as enumerable (IHDR, PLTE, tRNS, ..), and add further data fields. The whole PNG file is specified last (lines 32-42). It starts with a specific magic number

(Signature for PNG files), followed by a header chunk (IHDR) and upto 30,000 chunks (in flexible order) before ending up with an IEND chunk.

```
<DataModel name="Chunk">
  <Number name="Length" size="32" >
    <Relation type="size" of="Data" />
  </Number>
  <Block name="TypeData">
    <Blob name="Type" length="4" />
    <Blob name="Data" />
  </Block>
  <Number name="crc" size="32" >
    <Fixup class="Crc32Fixup">
      <Param name="ref" value="TypeData"/>
    </Fixup>
  </Number>
</DataModel>
<DataModel name="Chunk_IHDR" ref="Chunk">
  <Block name="TypeData">
    <String name="Type" value="IHDR" />
    <Block name="Data">
      <Number name="width" size="32" />
      <Number name="height" size="32" />
      ...
    </Block>
  </Block>
</DataModel>
...
<DataModel name="Chunk_tRNS" ref="Chunk">
  <Block name="TypeData">
    <String name="Type" value="tRNS" />
    <Blob name="Data" />
  </Block>
</DataModel>
<DataModel name="PNG">
  <Number name="Sig" value="89504e..." />
  <Block name="IHDR" ref="Chunk_IHDR"/>
  <Choice name="Chunks" maxOccurs="30000">
    <Block name="PLTE" ref="Chunk_PLTE"/>
    ...
    <Block name="tRNS" ref="Chunk_tRNS"/>
    <Block name="IDAT" ref="Chunk_IDAT"/>
  </Choice>
  <Block name="IEND" ref="Chunk_IEND"/>
</DataModel>
```

Listing 5.2: PNG input model as Peach Pit

Given the setup, to generate the crashing input in the motivating example, MoBWF manages to (i) insert a tRNS chunk into proper position in a benign PNG file, (ii) explore the paths affected by the existence of tRNS towards crash location, and (iii) generate specific value for the image width data field in IHDR chunk. This is achieved in four steps.

Step 1. Seed selection and file cracking. As shown in Figure 5-2, MoBWF first selects as *initial input* that file which is closest to a potential crash location. All other PNG files are considered donors, disassembled by the *file cracker* and added to the *fragment pool*. File fragments can be transplanted into input files as needed. If no initial files are provided, MoBWF instantiates the initial input from the input model. Then, MoBWF marks as *symbolic* all data fields which the user specified as "modifiable". Only modifieable data fields are considered for the fuzzing. In this example, all data fields (e.g., image width) are marked as modifiable except for the chunk's checksum and size. The resulting hybrid symbolic PNG file (i.e., some parts are symbolic where others are concrete) is then executed concolically by a traditional whitebox fuzzer.

Step 2. Adding and removing data chunks. Certain branches in a file-processing program are exercised only if a certain *data chunk* is absent or present. To exercise these branches during path exploration, MoBWF removes the specific chunk or adds a new one. First, in the execution of a given file f, MoBWF identifies those *crucial if-statements* (IFs) by their dependence on a data field in f of enumerable type. In Listing 5.1, the IFs in lines 11–26 can be considered crucial while none of the those inside the `handle_****` functions are. In our experiments, we observe that such enumerables do often uniquely identify a data chunk's type. First, MoBWF identifies the input bytes in f that influence the

outcome of executed branch predicates using classical *taint analysis*. In our example, MoBWF determines the relationship between the input bytes above the grey boxes in Figure 5-1 and the IFs in Listing 5.1. Then, MoBWF learns the type of the referenced data field using the input model. Finally, if the data field is of enumerable type and the IF is not already executed in both directions, then the IF is considered crucial and MoBWF removes the corresponding data chunk or adds a new one through transplantation or instantiation from the input model.

Once MoBWF identifies the type corresponding to the data chunk being removed or added, the *file stitcher* coordinates the data chunk transplantation. First, the stitcher searches the fragment pool for candidate data chunks that are allowed (according to the input model) to be put at the same level as the chosen chunk in the current seed file f. Finally, the file sticher uses the input model to identify the set of input bytes corresponding to each candidate data chunk in the pool and transplants them into the appropriate location of the receiving file f to generate a number of new seed files, one for each chunk. For our example, in what follows we assume that the candidate containing the tRNS chunk is chosen next.

Step 3. Changing data fields in inserted data chunk. Other branches in a file-processing program are exercised only if specific values are set in the chunks' *data fields*. In our example, the vulnerability is exposed only when the image width is in a range of certain values. To exercise these branches by finding the specific values is the strength of whitebox fuzzing. Selective symbolic execution explores the local search space of semi-valid inputs starting from the negated crucial branch. This local search is very efficient when compared to classical TWF. During exploration, any integrity check is identified and ignored. The potentially

invalid files are later fixed during the *file repair*. Once the target location is reached, the whitebox fuzzer checks the satisfiability of the conjunction of path constraint and crash condition (inferred from the given crash report or provided as output of static analysis tool). If the conjunction is satisfiable, the whitebox fuzzer generates a crashing input. Otherwise, it uses the unsatisfiable core to guide the path exploration towards the crash location and does the check again.

Step 4. Repeat. Data chunks can be nested in certain file formats (such as WAV). Thus, MoBWF uses the generated files as new seeds to continue the next iteration starting from Step 1. From the augmented seeds (initial seeds + new seeds), MoBWF selects a file which is closest to the crash location and moves to next steps. MoBWF executes selected file, identifies crucial if-statements, transplants data chunks and continues path explorations.

Summary. In this motivating example, MoBWF follows these four steps. During concolic execution, it identifies line 25 (Listing 5.1) as crucial if-statement. From the input model, the *file stitcher* infers that a tRNS chunk is a candidate for transplantation and it is allowed after PLTE and before the IDAT chunk. So, *file stitcher* transplants a tRNS chunk from the fragment pool or directly instantiates a minimal tRNS chunk from the input model and places it right before IDAT chunk. As a result, the true branch of the if-statement in line 25 is taken and the tRNS chunk is parsed before doing further processing. Once the crash location is reached, the image-width dependent crash condition is checked and a PNG file is produced. The resulting file is still invalid because the new value of image width invalidates the checksum of IHDR chunk. So, the file repair tool fixes the checksum and the vulnerability is exposed.

5.3 Model-based Black-box and White-box Fuzz Testing

Algorithm 2 gives an overview of the procedure of MoBWF. It takes a program $\mathcal{P}$, an input model $\mathcal{M}$, a set of target locations L in $\mathcal{P}$, and seed inputs T. The *objective* of Algorithm 2 is to generate valid (crashing) files that exercise L. If no target is provided, MoBWF uses static analysis to identify dangerous locations in the program, such as locations for potential null pointer dereferences or divisions by zero (line 1-2). The algorithm uses the provided test cases T as seed inputs for the test generation. However, if no seed file is provided, MoBWF leverages the input model $\mathcal{M}$ to instantiate a seed file (lines 3-5).

Algorithm 2 Model-based Blackbox and Whitebox Fuzzing

Input: Program $\mathcal{P}$, Input Model $\mathcal{M}$
Input: Initial Test Suite T, Targets L
Output: Augmented Test Suite T'

1: **if** $L = \emptyset$ **then**
2: $L \leftarrow$ identifyCriticalLocations($\mathcal{P}$)
3: **if** $T = \emptyset$ **then**
4: $t \leftarrow$ instantiateAsValidInput($\mathcal{M}$)
5: $T \leftarrow \{t\}$
6: **while** timeout not exceeded **do**
7: Target location $l \leftarrow$ chooseTarget(L)
8: Input file $t \leftarrow$ chooseBest(T, l)
9: Fragment Pool $\Phi \leftarrow$ fileCracker($T, \mathcal{M}$)
10: Crucial ifs $\Lambda \leftarrow$ detectCrucialIFS($t, l, \mathcal{P}, \mathcal{M}$)
11: **for all** $\lambda \in \Lambda$ **do**
12: Valid files $T_\lambda \leftarrow$ fileStitcher($t, \lambda, \Phi, \mathcal{M}$)
13: **for all** $t_\lambda \in T_\lambda$ that negate λ **do**
14: Hybrid file $\hat{t}_\lambda \leftarrow$ markSymbolicVars($t_\lambda, \mathcal{M}$)
15: Files $F \leftarrow$ pathExploration($\hat{t}_\lambda, \lambda, l, L, \mathcal{P}$)
16: **for all** $f \in F$ **do**
17: Valid file $f' \leftarrow$ fileRepair($f, \mathcal{M}$)
18: $T \leftarrow T \cup f'$
19: $T' \leftarrow T$

The main loop of Algorithm 2 is shown in lines 6-18. First, MoBWF

chooses the next target location l. If MoBWF works in crash reproduction mode, l is the known crash location extracted from the given crash report. Otherwise, l is picked if its average distance to all seed inputs in T is smallest. The distance between an input t and a program location l is specified in Definition 1. Second, MoBWF chooses the next seed file t according to a search strategy that seeks to generate the next input with a reduced distance to l (line 8). The remaining seed files are sent to the file cracker to construct the fragment pool Φ in line 9. The fragment pool takes a central role during data chunk transplantation.

Definition 1 (Input Distance to Location)**.** *Given an input t, a program $\mathcal{P}$ and a program location l in $\mathcal{P}$. Let $\Omega(t)$ be the set of nodes in the Control Flow Graph (CFG) of $\mathcal{P}$ that are exercised by t. The* distance $\delta(t, l)$ *from t to l is the number of nodes on the shortest path from any $b \in \Omega(t)$ to l.*

Next, Algorithm 2 executes t on $\mathcal{P}$ to determine crucial IFs Λ (line 10). As specified in Definition 2, a crucial IF is evaluated in different directions only depending on the type of the data chunks present in t. Our implementation leverages $\mathcal{M}$ to identify crucial IFs by their dependence on a data field in t of enumerable type. We observed that such enumerables do often uniquely identify a data chunk's type. Note that we ignore executed IFs negating which does not reduce the distance to the target location l.

Definition 2 (Crucial IF-statement)**.** *Given input t for program $\mathcal{P}$ and a target location l in $\mathcal{P}$, an if-statement b in $\mathcal{P}$ is* crucial *if*

1) the statement b is executed by t in $\mathcal{P}$,

2) only one direction of b has been taken,

3) the negation of the branch condition at b reduces the distance to l, and

4) let $\varphi(b)$ be the branch condition at b; the outcome of $\varphi(b)$ depends on a

field in t that specifies the chunk's type.

For each crucial IF λ thus identified, Algorithm 2 employs the file stitcher to negate λ's branch condition (lines 11-12). For each stitched file t_λ that successfully negates λ, the algorithm executes selective symbolic execution followed by file repair to fine-tune the specific values of the data chunks and reduce the distance to l (lines 13-18). More specifically, it marks all modifiable data fields in t_λ as symbolic and starts the directed path exploration (lines 14-15). During path exploration, MoBWF does not collect integrity checks as branch constraints. For instance, a checksum check might not allow to change a data field which would otherwise lead to reducing the distance to L (cf. TaintScope [101]). Such integrity constraints are repaired in line 17. Whenever a potential dangerous location in L is reached, MoBWF checks if the crash condition is satisfied and generates a crashing test case accordingly.

5.3.1 Directed Model-based Search

In order to generate inputs that expose vulnerabilities, MoBWF uses the initial seed inputs T to reduce the distance to the provided or identified critical location l until it is reached and the crash condition is satisfied.

Critical Locations. If no targets L are provided to the algorithm, MoBWF identifies critical locations in the program $\mathcal{P}$. A *critical location* is a program location that may expose a vulnerability if exercised by an appropriate input. There are several methods to identify such critical locations [45, 101]. In our implementation, we use IDAPro [7] to dissamble the program binary $\mathcal{P}$ and perform some lightweight analysis to identify instructions that conform to the patterns shown in Listing 5.3. These patterns partially cover program instructions that may trigger divide-by-zero and null-pointer dereference vulnerabilities. Specifically, we

focus on division and memory move instructions taking registers or stack arguments as operands. For those instructions, the crash condition is obvious. Once a critical location is reached during concolic exploration, we just check whether the value of register/stack argument is zero (in case it is concrete) or can be zero (in case it is symbolic).

```
div   register
div   [ebp + argument_offset]
mov   operand, [register]
mov   operand, [ebp + argument_offset]
mov   [register], operand
mov   [ebp + argument_offset], operand
```

Listing 5.3: Crash instruction templates

Model-based Search. To generate input that reduces the distance to l, MoBWF first chooses the seed input t with the least distance to l and then identifies the executed crucial IFs Λ (lines 8, 10 in Alg. 2). The task of the subsequent data chunk transplantation and instantiation will be to generate valid inputs that negate the branch conditions of Λ. While other implementations are possible, we decided to implement a hill climbing algorithm. Our implementation of CHOOSEBEST selects the input file $t \in T$ such that for selected location $l \in L$ we have that the distance from t to l is minimal. To detect crucial branches Λ, MoBWF first determines, using taint analysis, those input bytes in t that may impact the outcome of some $b \in \Omega(t)$. We recall that $\Omega(t)$ is the set of nodes in the CFG of program $\mathcal{P}$ which are exercised by t. In our implementation of DETECTCRUCIALIFS, we leverage those capabilities in a symbolic execution tool, Hercules. Next, MoBWF uses the CFG to compute the number of nodes on the shortest path between b and location $l \in L$. The negation of $\varphi(b)$ may reduce the distance to l only if b is in static backward slice of l and the branch b'

immediately following b does not have a smaller number of nodes on the shortest path between b' and l. Lastly, MoBWF uses $\mathcal{M}$ to determine the data field corresponding to the identified input bytes and whether the data field specifies the chunk's type. If all conditions specified in Definition 2 are met, then b is marked as a crucial IF and added to Λ.

5.3.2 Transplantation, Instantiation, and Repair

File Cracker. "File cracking" refers to the process of interpreting valid files according to a provided input model (i.e., the Peach Pit file). Given the input model $\mathcal{M}$ and a valid file $t \in T$, the FILECRACKER identifies all data chunks and their data fields in t. In model-based blackbox fuzzers like Peach Fuzzer [9], the valid input files are cracked and fuzzed independently. However, in MoBWF we crack all files and place their data components inside a *fragment pool*. As a result, we can consider all files (and even the input model) as donors for data transplantation. By doing that, MoBWF can generate more (semi) valid files and improve coverage.

File Stitcher. Given a valid file t and the crucial IF λ, the objective of FILESTITCHER is to negate $\varphi(\lambda)$ and reduce the distance to l by adding or removing chunks from t. First, the stitcher has to determine the chunk c in t that should be removed or before which a different chunk should be added in order to negate $\varphi(\lambda)$. Chunk c was memorized previously when determining that the outcome of λ depends on the data field specifying c's type. Second, the stitcher generates a new file by removing c from t if allowed according to $\mathcal{M}$. Third, for each chunk type $\mathcal{C}$ that is allowed before c in t:
i) *Transplantation.* If there exists a chunk c' of type $\mathcal{C}$ in the pool Φ, copy the input bytes corresponding to c' from the donor file to the position before c in the receiving file t.

ii) *Instantiation*. Otherwise, use the specification of $\mathcal{C}$ in $\mathcal{M}$ as a template to generate the bytes for c' before c in t. All files thus generated that actually negate λ will be used for the subsequent selective symbolic execution stage.

File Repair. Given a file f and the input model $\mathcal{M}$, the file repair tool re-establishes the integrity of the file. Our implementation utilizes the fixup and transformers that can be specified in $\mathcal{M}$ in the Peach framework.

5.3.3 Selective and Targeted Symbolic Execution

We reuse the targeted search strategy for symbolic exploration implemented in Hercules [88]. Basically, to mitigate the path explosion problem, it enables fully symbolic reasoning only in some selected modules of interest (i.e., executable binaries like .exe and .dll files). The list of selected modules can be inferred from the target module TM, which contains the selected target location, and a so-called Module Dependency Graph (MDG). The MDG is constructed by running the program under test with benign inputs and collecting the control transfer between program modules. Using the constructed MDG, TM and all modules on paths from entry module (main program) to TM are selected to explore in fully symbolic execution mode.

The search strategy of Hercules is targeted in the sense that it explores program paths towards a target location (critical locations like crashing one) by pruning irrelevant paths. Moreover, Hercules leverages the unsatisfiable core produced by a theory prover like Z3 [40] to guide the exploration.

5.3.4 Handling Incomplete Memory Modeling

The memory models of symbolic execution engines, like Hercules, KLEE or S2E [88, 28, 35], do not support memory allocation with symbolic size. If a symbolic size is given, it is concretized before allocating heap memory. The concretization mechanism could prevent us from exposing heap buffer overflow vulnerabilities. Suppose in the motivating example the image width of the benign PNG file is very small, say 1, and it is marked as symbolic. In the processing code, LibPNG needs to allocate a heap buffer having symbolic size that depends on *width* (and other symbolic variables). When the buffer is allocated, *width* is bound in PC by the constraint on concretized value for allocated buffer size.

Once the crash location (e.g., the instruction accessing the allocated heap buffer) is reached, Hercules checks the satisfiablity of the conjunction between the current path constraint PC and the crash condition CC. Suppose that to satisfy the crash condition, the image width must be large enough. For the current file with the small image width, the crash condition CC could contradict the path constraint PC; $PC \wedge CC$ is unsatisfiable. Usually, based on the unsatisfiable core[1] of $PC \wedge CC$, Hercules find a set of branches that can be negated to explore neighboring paths along which the crash condition CC may be satisfiable. However, since *width* is already bound, there exists no alternative path along which the crash condition CC can be satisfied.

In our extension of Hercules, we leverage recent advances in maximal satisfication with Z3 (MaxSMT)[19, 40]. MaxSMT allows us to select a subset of constraints which is not required to be satisfied as "soft constraints" while the remaining constrains (which need to be satisfied)

[1]Given an unsatisfiable Boolean propositional formula in conjunctive normal form, a minimal subset of clauses whose conjunction is still unsatisfiable is called an unsatisfiable core of the original formula.

are implicitly marked as "hard constraints". Specifically, in our case we set all constraints in CC as hard constraints while specifying e.g., constraints due to memory allocation in PC as soft constraints. To identify which constraints in PC can be soft, first we check whether the conjunction $PC \wedge CC$ is unsatisfiable. If so, we extract all symbolic variables in CC. Thereafter, we iterate through all constraints in PC and consider them as soft constraints accordingly if they contain any symbolic variable from CC. After all these steps, we get PC', the updated PC, and we send another query to MaxSMT solver to check the maximum satisfiability of $PC' \wedge CC$. If $PC' \wedge CC$ is satisfiable (by possibly making one or more soft constraints in PC' as false) – we generate a input file as the solution to the constraints. As an additional confirmation, we validate the generated file by feeding it to the program binary and checking whether it crashes the program.

5.4 Implementation

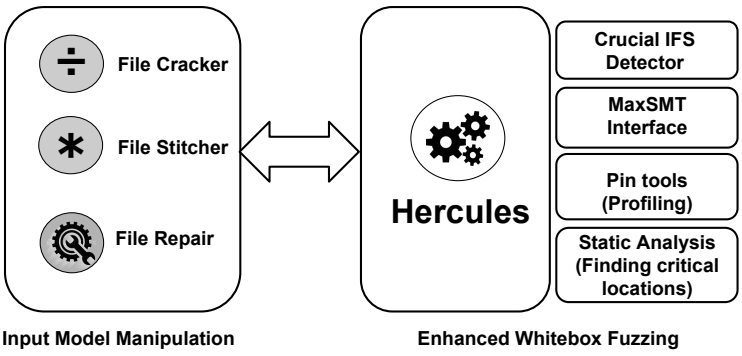


Figure 5-3: Components of our MoBWF tool

Our MoBWF tool is based on several third-party tools and libraries. We implemented our technique into the `Hercules` [88] directed symbolic execution engine which itself leverages `S2E` [35] and the `Z3` [40] satisfiability modulo theory constraint solver. We also improved the accuracy of the

taint analysis that is implemented in `Hercules`. `IDAPro` [7] and the Intel Dynamic Binary Instrumentation Tool [79] (or `PIN tool`) were used for static analysis to find dangerous locations in the program code executing The `PIN tools` were also used i) for instruction profiling to generate the execution trace and compute the distance of the current seed input to the dangerous locations, and ii) for branch profiling to determine which crucial branches are explored. The framework around the `Peach` model-based blackbox fuzzer [9] allowed us to implement the input model-based components such as *File cracker*, *File Stitcher* and *File Repair*. In fact, the first was modified for our purposes and the latter two were implemented from scratch, for instance, to support data chunk transplantation.

5.5 Experimental Evaluation

We evaluated our MoBWF technique experimentally to answer the following research questions.

- **RQ.1** How many vulnerabilities are exposed by MoBWF compared to Traditional Whitebox Fuzzing (TWF)?

- **RQ.2** How many vulnerabilities are exposed by MoBWF compared to Model-based Blackbox Fuzzing (MoBF)?

- **RQ.3** How many vulnerabilities are exposed by MoBWF if no initial seed inputs are available?

Each technique was evaluated with a 24 hour time budget.

Program	Version	Buggy module	Size	Errors
Video Lan Client	2.0.7	libpng.dll	184 KB	1
Video Lan Client	2.0.3	libpng.dll	182 KB	1
Libpng Test Program	1.5.4	libpng.dll	176 KB	1
XnView	1.98	XnView.exe	4.46 MB	0 + 3
Adobe Reader	9.2	cooltype.dll	2.32 MB	1
Windows Media Player	9.0	quartz.dll	1.22 MB	2 + 1
Real Player SP	1.0	realplay.exe	60 KB	1
MIDI Player	0.35	mamplayer.exe	336 KB	1
Orbital Viewer	1.04	ov.exe	538 KB	1
			Total:	9 + 4

Table 5.1: Subject Programs

5.5.1 Experimental Setup

Subjects

We selected our subjects from a pool of well-known program binaries of video players, document readers, music players, and image editors – which take a variety of complex file formats. Since Hercules serves as a base line technique, we also added all five subjects on which Hercules was evaluated originally [88] (shown with grey background). We also took the categories of vulnerabilities into consideration. As shown in Table 5.1, we chose eight distinct real-world applications (some with different versions): Adobe Reader (AR)[2], Video Lan Client (VLC)[3], Windows Media Player (WMP), Real Player (RP)[4] and Music Animation Machine MIDI Player (MP)[5], XnView (XNV)[6], LibPNG (LTP)[7] and Orbital Viewer (OV)[8].

Table 5.1 shows not only the subjects and their versions but also the target buggy modules and their respective sizes. In addition, it features the number of known vulnerabilities that we sought to reproduce. In one

[2]https://get.adobe.com/reader/
[3]http://www.videolan.org/index.html
[4]http://www.real.com/sg
[5]http://www.musanim.com/player/
[6]http://www.xnview.com/en/
[7]http://www.libpng.org/pub/png/libpng.html
[8]http://www.orbitals.com/orb/ov.htm

case (XnView), we started without any known vulnerabilities and looked for unknown ones. In other cases, although we targeted the known vulnerabilities, we managed to discover new ones. Indeed, our MoBWF tool reproduced successfully all 9 known errors and discovered 4 unknown errors – 3 in XnView and 1 in Windows Media Player (See Section 5.5.2).

Input Modeling

To *define input models* of five file formats (PDF, PNG, MIDI, FLV and ORB) from scratch, we utilized the modeling language of the Peach model-based blackbox fuzzer. We augmented the input model for WAV files which is provided freely by Peach Fuzzer. In particular, we modeled one common image file (PNG), three audio and video files (MIDI, WAV and FLV), one portable document file (PDF) and one geometry file (ORB). In Table 5.2, we report the size of the input models which are relatively small – ranging from 4 KB to 14 KB. It took us less than a day to write each model for a file format.

Format	**Size**	**Time spent**	**#Files**	**Average size**
PDF	4.5 KB	12 hours	10	200 KB
PNG	8.3 KB	4 hours	10	55 KB
MIDI	13.9 KB	4 hours	10	20 KB
FLV	6.0 KB	4 hours	10	300 KB
ORB	6.0 KB	8 hours	10	4 KB
WAV*	7.5 KB	2 hours	10	260 KB

Table 5.2: Information on the Input Models

Initial seed files selection

To *select the initial seed files*, we randomly downloaded 10 files of the corresponding format from the Internet, except ORB and PNG initial seed files. The ORB files were downloaded from software vendor's website[9] while

[9]http://www.orbitals.com/orb/ov.htm

PNG files were downloaded from the Schaik online test suite.[10] The average size of seed files in each test suite is shown in the fifth column of Table 5.2.

Infrastructure

We evaluated three tools, our MoBWF tool, the `Hercules` Traditional Whitebox Fuzzer (TWF) and the `Peach` Model-based Blackbox Fuzzer (MoBF). For the experiments, we used the community version of Peach Fuzzer which is provided with its source code.[11] Both model-based techniques used the same input models. All subject programs were run on Windows XP 32-bit SP 3. For each program, each tool was configured for a timeout after 24 hours of execution. We conducted all experiments on a computer with a 3.6 GHz Intel Core i7-4790 CPU and 16 GB of RAM.

5.5.2 Results and Analysis

Program	Advisory ID	Model	Files	MoBWF	MoBF	TWF
VLC 2.0.7	OSVDB-95632	PNG	10	✔	✘	✘
VLC 2.0.3	CVE-2012-5470	PNG	10	✔	✘	✘
LTP 1.5.4	CVE-2011-3328	PNG	10	✔	✘	✘
XNV 1.98	Unknown-1	PNG	10	✔	✔	✘
XNV 1.98	Unknown-2	PNG	10	✔	✔	✘
XNV 1.98	Unknown-3	PNG	10	✔	✔	✘
WMP 9.0	Unknown-4	WAV	10	✔	✔	✘
WMP 9.0	CVE-2014-2671	WAV	10	✔	✘	✔
WMP 9.0	CVE-2010-0718	MIDI	10	✔	✘	✔
AR 9.2	CVE-2010-2204	PDF	10	✔	✘	✔
RP 1.0	CVE-2010-3000	FLV	10	✔	✘	✔
MP 0.35	CVE-2011-0502	MIDI	10	✔	✔	✔
OV 1.04	CVE-2010-0688	ORB	10	✔	✔	✔

Table 5.3: The vulnerabilities exposed by our MoBWF tool, the `Hercules` TWF, and the `Peach` MoBF. Vulnerabilities from the `Hercules` benchmark are marked as grey.

Table 5.3 shows the results in reproducing known vulnerabilities and

[10] http://www.schaik.com/pngsuite
[11] http://community.peachfuzzer.com to download.

finding unknown ones of the three compared techniques. Overall, in the experiments our MoBWF tool outperforms both `Hercules` and `Peach`. While our MoBWF tool successfully generated 13 crash-inducing inputs, neither `Hercules` nor `Peach` can produce half of them . Furthermore, our MoBWF tool also found potential unknown vulnerabilities in Windows Media Player and XnView. Indeed, these vulnerabilities have previously not been reported at MITRE[12], OSVDB[13] or Exploit-DB.[14]. In addition, the power of our MoBWF tool is also demonstrated by its ability to expose different types of vulnerabilities including integer and buffer overflows, null pointer dereference and divide-by-zero. In the following sections, we have an in-depth analysis to answer the three research questions about the effectiveness and sensitivity of our approach.

RQ.1 Versus Traditional Whitebox Fuzzing

Our experiments confirm the observations that TWF is unlikely to synthesize missing composite data chunks. As in OSVDB-95632, CVE-2012-5470, CVE-2011-3328 and Unknown 1-4, `Hercules` cannot produce crash inputs to expose the vulnerabilities because they require the existence of optional composite data chunks. In our experiments, `Hercules` gets stuck in synthesizing such required data chunks. In particular, the following requirments must be met to expose the 7 vulnerabilities that are not in the `Hercules` benchmark:

OSVDB-95632 (Buffer Overflow): It requires a PNG file with a tRNS optional data chunk specifing either alpha values that are associated with palette entries (for indexed-colour images) or a single transparent colour (for greyscale and truecolour images). Moreover, the value of a data

[12]http://cve.mitre.org/
[13]http://osvdb.org/
[14]https://www.exploit-db.com/

field (image width) in IHDR chunk (the header chunk of PNG) must be able to trigger an integer overflow in the LibPNG plugin in VLC 2.0.7.

CVE-2012-5470 (Buffer Overflow): It requires a PNG file with a tEXt optional data chunk which stores text strings associated with the image, such as an image description or copyright notice. Furthermore, the length of the data chunk must be big enough to exceed the size of a heap buffer allocated for the image. However, it cannot be so huge that it prevents LibPNG from successfully allocating a heap buffer that is supposed to store the data in tEXt chunk.

CVE-2011-3328 (Divide-by-Zero): They require a PNG file with a cHRM optional data chunk. The cHRM specifies chromaticities of the red, green, and blue display primaries used in the image, and the referenced white point. Second, some data fields in cHRM chunk must have specific values to trigger a divide-by-zero bug in the LibPNG library.

Unknown 1-3 (Memory Read Access Violation): They require PNG files having optional data chunks (iTXt, zTXt or iCCP accordingly) which have no content. That is, the chunks that specify a size of zero followed by chunk name and checksum.

Unknown 4 (Divide-by-Zero): It requires a WAV file in which the format chunk contains an optional extra composite data field and one specific byte in the field is zero.

Unlike `Hercules`, our MoBWF tool leverages the input models to transplant required data chunks from other files in the initial test suite or generate the chunks automatically from the input model. Hence, our MoBWF tool can successfully produce crash inputs as witnesses for the seven vulnerabilites mentioned above.

Since our MoBWF tool is an extension of `Hercules`, it can successfully reproduce all six vulnerabilites in the `Hercules` benchmark. As we will see

for RQ.3, our MoBWF tool does not require seed inputs to reproduce three out of the six vulnerabilites in the `Hercules` benchmark (CVE-2010-0718, CVE-2011-0502 and CVE-2010-0688) because of its capability to generate (semi-) valid files directly from input models.

RQ2. Versus Model-based Blackbox Fuzzing

The `Peach` model-based blackbox fuzzer cannot expose half of the vulnerabilities that our MoBWF tool can expose (see Table 5.3). We note that we conservatively assume that data chunk transplanation and instantiation is available in `Peach` – even though it is not. It is worth mentioning that supporting transplanation and instantiation in `Peach` could be challenging. In fact, finding the correct chunk to transplant and transplanting it to the correct location in the seed input is subject to combinatorial explosion in an undirected fuzzing technique like `Peach`. In constrast, MoBWF uses information about crucial IFs to direct the transplantation.

In the experiments, we simulated `Peach`'s capability to do data chunk transplanation and instantiation by augmenting the set of all 10 seed inputs where none contains the missing data chunk with at least one seed input where we manually transplanted the missing data chunk. In Table 5.3, we indicate that `Peach` (with the simulated capability) can expose three vulnerabilities Unknown 1-3 since these only require the existence of empty-data optional chunks.

However, for the remaining 10 vulnerabilities, the MoBF tool `Peach` cannot successfully expose 7 of 10 vulnerabilities even though we provide inputs with the required optional data chunks. It is because of its *limitation on generating specific values*. The reason lies with the inability of blackbox fuzzing to generate the specific values for data fields that would expose

deep vulnerabilities. For example, given a 4-byte integer data field, the chance for a blackbox fuzzer to randomly mutate and get a specific value X is extremely small, just only $1/2^{32}$. In contrast, symbolic execution-based whitebox fuzzing is very good at finding such values.

Meanwhile, our MoBWF tool is an enhancement of TWF (by leveraging input models) and can tackle both the missing data chunk problem and the limitation on generating specific input values. As a result, it can successfully produce test cases to expose all of the 13 vulnerabilities.

RQ3. Sensitivity to the initial test suite

Program	**Advisory ID**	**Model**	**#Files**	**MoBWF**
VLC 2.0.7	OSVDB-95632	PNG	0	✔
VLC 2.0.3	CVE-2012-5470	PNG	0	✔
LTP 1.5.4	CVE-2011-3328	PNG	0	✔
XNV 1.98	Unknown-1	PNG	0	✔
XNV 1.98	Unknown-2	PNG	0	✔
XNV 1.98	Unknown-3	PNG	0	✔
WMP 9.0	Unknown-4	WAV	0	✘
WMP 9.0	CVE-2014-2671	WAV	0	✘
WMP 9.0	CVE-2010-0718	MIDI	0	✔
AR 9.2	CVE-2010-2204	PDF	0	✘
RP 1.0	CVE-2010-3000	FLV	0	✘
MP 0.35	CVE-2011-0502	MIDI	0	✔
OV 1.04	CVE-2010-0688	ORB	0	✔

Table 5.4: Vulnerabilities exposed by our MoBWF tool if no initial seed files are provided.

For this experiment, we run our MoBWF tool with no initial seed inputs as shown in Table 5.4. By leveraging input models of PNG, MIDI and ORB, for each file format our MoBWF automatically generates one minimal seed file. In particular, a minimal PNG file is an 1x1 image having four mandatory chunks – IHDR, PLTE, IDAT and IEND. In case of MIDI, it is a single track audio file with one header chunk (MThd) and one audio track chunk (MTrk). The minimal ORB file contains all required properties

for rendering an orbital object. Once the files are generated, we run our MoBWF tool on all subjects listed in Table 5.4.

The experiments show that with the minimal files, our MoBWF tool can expose 9 of 13 vulnerabilities (which can be revealed by PNG, MIDI and ORB files) as reported in Table 5.3. It means that our MoBWF tool exposes 70% vulnerabilites without any provided seed inputs providing evidence that MoBWF technique reduces the dependence of TWF on selected seed inputs.

MoBWF does not succeed in exposing the vulnerabilities in 4 of 13 vulnerabilities because they require WAV, FLV and PDF files as inputs. However, our models for these file formats are still coarse. Although they are enough to allow MoBWF to work with given test suites, they need to be more complete to support directly generating (semi-) valid files. Since these file formats are complex, on one hand we can spend more time to read and fully understand their specifications in order to augment the input models. On the other hand, we can reuse exhaustive models written by software vendors or the owners of file formats. For instance, according to a post at the official Adobe Blog,[15] developers at Adobe System wrote their model for PDF file (which was a proprietary format controlled by Adobe until 2008) and used Peach Fuzzer to fuzz their most popular software – Adobe Reader. Given such (partially) complete input models, our MoBWF approach would complement MoBF tool like Peach Fuzzer to maximize the utility of these models and hence expose more vulnerabilities.

[15]https://blogs.adobe.com/security/tag/fuzzing

5.6 Threats to Validity

The main threat to external validity is the generality of our results. MoBWF has been developed for real-world program binaries that take complex program inputs. We choose a variety of well-known programs from different domains where specifications of the input models are available. While for proprietary applications such format specifications might not be available, we believe that grammar inference techniques can be a powerful tool to automatically derive the input model. Half of the vulnerabilities have already been picked in earlier work [88]. To showcase the effectiveness of MoBWF, the other half has been chosen such that an optional data chunk is required to expose the vulnerability.[16]

The main threat to internal validity is selection bias during the seed selection (see Table 5.2). We chose the seed inputs either randomly from a benchmark or from the internet. Moreover, our experiments confirm the reduced dependence on the available seed inputs.

The main threat to construct validity is the correctness of our implementation. However, our tool is an extension of both Hercules and Peach, the two baselines for our evaluation. So, our tool inherits the incorrectness of the baseline.

5.7 Chapter Summary

In this chapter, we introduced MoBWF as an automated testing technique for program binaries that process highly structured inputs. We have observed that certain branches in a file-processing program are exercised only depending on i) the *presence* of a specific data chunk, ii) a *specific value* of a data field in a data chunk, or iii) the *integrity* of the

[16]See RQ.1. in Section 5.5.2.

data chunks. Hence, we extend HERCULES an existing traditional whitebox fuzzing technique not only to set specific values of the fields but also to add/remove complete chunks and re-establish their integrity during fuzzing.

MoBWF is a promising fuzzing technique for program binaries that process highly structured input. It is particularly helpful when no initial seed files are available that contain the required optional data chunks. Given the same time budget, MoBWF can generate more valid test inputs which aids in exposing vulnerabilities that could not be exposed otherwise.

Chapter 6

Directed Coverage-based Grey-box Fuzz Testing

Coverage-based Greybox Fuzzing (CGF) has shown its effectiveness in discovering numerous vulnerabilities reported today. However, given a specific set of target locations, say the methods in a stacktrace of an in-field crash that an in-house developer wishes to reproduce or updated functions in a new code commit which should be thoroughly tested to prevent regression bugs, CGF cannot be directed towards quickly generating seeds that can reach these targets. In this chapter, we present our approach to integrating the capability to be directed by a set of targets into CGF.

6.1 Introduction

Coverage-based Greybox Fuzzing (CGF) is a random testing approach where new program inputs are generated by slightly mutating a seed input: If the input exercises a new branch (which is not covered by the existing seeds), it is added to the set of seeds. Light-weight instrumentation allows to check for an increase in coverage with close to

no overhead. CGF (as implemented in several popular fuzzing tools like AFL and LibFuzzer [4, 8]) is a powerful automated vulnerability detection technique, perhaps because it is both scalable as well as highly parallelizable. It is *scalable* because the time to generate a test does not increase with the program size and *highly parallelizable* because the retained seeds represent the only internal state. Several CGF instances can be run in parallel with a shared queue. A *shared queue* allows one instance to access all the seeds that have been discovered by any other instance. However, given a specific set of target locations, say the methods in a stack trace of an in-field crash that an in-house developer wishes to reproduce or updated functions in a new code commit which should be thoroughly tested to prevent regression bugs, CGF cannot be used to progressively reach these targets.

In this work, we augment CGF and make it directed towards a given set of targets by integrating into it a global search algorithm. We leverage the observation that CGF can be modeled as a Markov chain which specifies the probability p_{ij} that fuzzing the seed which exercises path i generates an input that exercises path j. In the case of AFL, j might be a path that would exercise branch that has not been covered. Böhme et al. [22] introduce so-called power schedules to effectively navigate the Markov chain. A *power schedule* assigns energy to each seed according to some function. The *energy* of a seed determines how many inputs are generated from that seed the next time it is chosen for fuzzing. Böhme et al. developed several power schedules that help to gravitate the fuzzer towards low-frequency paths rather than "wasting" energy on high-frequency ones. The fuzzer discovers more interesting paths per unit time.

This inspired us to integrate a well-known Markov Chain Monte Carlo

(MCMC) meta-heuristic into CGF by developing a suitable power schedule. This so-called temperature-based power schedule assigns energy depending on the seed's distance to the set of target locations. Specifically, we integrate *Simulated Annealing* (SA) as global search algorithm where a short distance to the set of targets becomes increasingly more important as time progresses. Intuitively, in the beginning almost every seed is assigned the same energy to allow initially for sufficient freedom to explore possibly less progressive paths. At a given point of time, which we call *time-to-exploitation*, the search enters the exploitation phase where seeds that are "closer" to the targets are assigned significantly more energy than those further away. Directed CGF is effectively a novel *single-objective, multi-target search-based software testing* technique.

We implemented the technique into AFL, which is the state-of-the-art of CGF, and call our tool AFLGO. We evaluated AFLGO as a crash reproduction tool on the stack traces of ten vulnerabilities in LibPNG and Binutils. Moreover, we also evaluated AFLGO as a patch testing tool for vulnerability detection on the changes in the commit that introduced the famous Heartbleed vulnerability [103] and on the changes in the 1600 most recent revisions of Binutils. Results are encouraging. AFLGO reproduced the vulnerabilities in LibPNG between three (3) and five (5) times faster than AFL and for those in Binutils usually about twice as fast. In patch testing mode, AFLGO exposed Heartbleed in less than six (6) hours while AFL took more than 20 hours. Notably, AFLGO discovered 14 zero-day vulnerabilities in Binutils of which three (3) vulnerabilities exist because of previous incomplete fixes. We filed bug reports for the discovered vulnerabilities and all of them have been confirmed and fixed by Binutils' maintainers. We also got five (5) CVEs

assigned to the most critical vulnerabilities.

This work makes the following contributions:

Path Distance. We develop a novel measure of *path distance* to a given set of targets. Path distance accounts for outlier targets that are far even from other targets. Moreover, it prefers paths that are closer to exercising one target but further from another over paths that are equi-distant from both targets.

Directed Fuzzing. We develop a Temperature-based Power Schedule (TPS) that integrates the efficiency of CGF and the directedness of Simulated Annealing global search algorithm. In our implementation, we take care that all program analysis that is required would be completed at compile time such that the overhead of our extension is negligible at runtime.

Multi-Target SBST. To the best of our knowledge, we develop the first multiple-target search-based software testing technique where the single objective is to generate an input that exercises as many of the *given* targets as possible. Previous work on Directed SBST is either on guidance towards a single target [104, 54, 82] or on the coverage of a maximal number of branches [43, 14].

AFLgo and Evaluation. We implemented directed CGF into AFL and evaluated AFLGO for the applications to crash reproduction and patch testing for vulnerability detection. In crash reproduction application, AFLGO normally exposes known vulnerabilities from two (2) to ten (10) times faster than AFL. AFLGO also shows its effectiveness in patch testing by discovering 14 zero-day vulnerabilities.

6.2 Motivating Example

We use the Heartbleed vulnerability to explain the pertinent properties of *directed coverage-based greybox fuzzing*—a light-weight system-level search-based testing generation technique. In this case, we direct the fuzzer towards the program locations that were changed in the commit that introduced the vulnerability. Our tool AFLGO implements the technique into the popular coverage-based greybox fuzzer AFL [4].

Heartbleed [103] (CVE-2014-0160, ♡) is a serious vulnerability that allows adversaries to decipher otherwise encrypted communication, for instance, during online banking. The vulnerability was accidently introduced into OpenSSL which implements the `https` protocol for secure communication and is used by the majority of servers on the internet. Heartbleed was introduced on Jan'12 when the "Heartbeats" feature was added. It was patched two years later in Apr'14. As of April 2016, a quarter million machines are still vulnerable [75]. One year after the patch, Böck showed how the fuzzer AFL could have found Heartbleed [21]. We decided to reuse his setup and see how much faster AFLGO could have found Heartbleed if it was run for the commit that introduced the vulnerability – merely directed towards the functions that had been changed. OpenSSL consists of more than *four thousand functions* out of which the commit that was supposed to add the Heartbeats-feature *changed twenty.*[1] Only after the commit one function contains a buffer overread which would become known as the Heartbleed vulnerability.

An overview of the AFLGO architecture is shown in Figure 6-1. When OpenSSL is compiled for AFLGO, the assembly-level instrumentation takes the targets (here, 20 changed functions) and adds a few assembly

[1]There are 4439 functions including library functions. We counted the number of nodes in the call graph lifted from the binary.

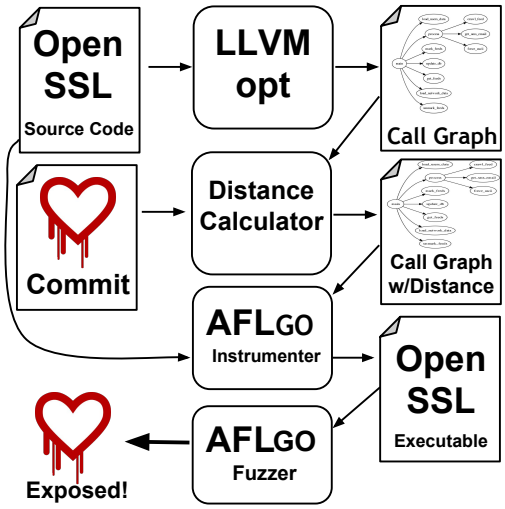


Figure 6-1: Overview AFLGO architecture.

instructions that indicate how "far" an executed seed input is from executing these targets. To maintain the fuzzer's efficiency, we wanted *no compromise at runtime*. So, *all program analysis is conducted at compile-time* such as a light-weight lifting of the call graph using the LLVM optimization tool `opt`. The distance calculator script marks the target functions and assigns to each node the distance to the targets. The node distance is computed using a *novel distance metric* that we introduce in this work. To the best of our knowledge, our metric seems to be the *first fitness function allowing for single-objective, multi-target SBST*. The AFLGO assembler then injects the computed distance into the existing trampoline.[2] While AFLGO instruments *during* compile time, is straight-forward to conduct this instrumentation *after* compilation [72].

When OpenSSL is fuzzed, the instrumented program informs AFLGO about the current distance of the seed to the targets. Instead of implementing a classical gradient descent and always preferring the seed with the least distance to the twenty changed functions, we implemented

[2]The *trampoline* is a piece of code injected by the AFL assembler that is executed after each jump instruction to keep track of the covered control-flow edges.

a global search that allows some exploration before gradually moving towards exploitation and finally degenerating to a classical gradient descent. In the *exploration* phase, a seed may be chosen even if it increases the distance to targets. In the *exploitation* phase, seeds with less distance are generally more preferred.

CVE	Fuzzer	Successful runs	μTTE	Factor
	AFLGo	30	5h41m	3.65
	AFL	18	20h46m	–

Figure 6-2: Improvement of AFLGo over AFL for Heartbleed.

In search-based software testing it is common to process individuals (i.e., seed inputs) in the order of their fitness, or to select only the fittest individuals. We take a different approach. We control the number of new individuals generated from one individual during fuzzing. Formally, we modify the *power schedule* of the fuzzer [22] rather than the order in which the fuzzer selects seeds from the queue. Moreover, unlike in search-based unit testing [42] where the goal is to generate a minimal sequence of method invocations to achieve a maximal coverage of a given unit (e.g., the SSL-object), AFLGo implements *search-based system testing* where the goal is to generate system-level inputs (e.g., for the public interface of OpenSSL) to quickly reach the given targets.

We implemented a so called temperature-based power schedule, which controls the number of new individuals generated from a seed differently. Then, we ran AFL and AFLGo with the power schedule on OpenSSL to measure the mean Time-To-Exposure (TTE) – the average time until the first seed is generated that exposes Heartbleed. We set a timeout for 24 hours, and repeated the experiment 30 times because fuzzing is essentially a random process. An unsuccessful run that did not expose Heartbleed in 24 hours is assigned a 24h TTE (rather than being unaccounted for). Figure 6-2 shows the number of successful runs, the mean TTE (μTTE)

and how much longer the average AFL run takes to expose the error versus the average AFLGO run (*Factor*).

The results are promising. If a continuous integration platform like Jenkins [73] had run AFLGO *for merely six hours as soon as the commit was submitted to the OpenSSL source code repository, then the vulnerability would have been found as it was introduced. AFL would have taken three to four times longer, almost a day.*

6.3 Background

6.3.1 Simulated Annealing

Simulated Annealing (SA) is a Markov Chain Monte Carlo (MCMC) method for approximating the global optimum in a very large, often discrete search space within an acceptable time budget [65]. The main feature of SA is that during the random walk it always accepts better solutions but sometimes it may also accept worse solutions. The *temperature* is a parameter of the SA algorithm that regulates the acceptance of worse solutions and is decreasing according to a cooling schedule. At the beginning, when $T = T_0$, the SA algorithm may accept worse solutions with high probability. Towards the end, when T is close to 0, it degenerates to a classical gradient descent algorithm and will accept only better solutions.

The simulated annealing algorithm *converges asymptotically* towards the set of global optimal solutions. This set of global optimal solutions, in our case, is the set of paths exercising the maximum number of targets. A *cooling schedule* controls the rate of convergence and is a function of the initial temperature $T_0 \in \mathbb{N}$ and the temperature cycle $k \in \mathbb{N}$. The *initial temperature* T_0 is provided and must be high enough so that any new solution is accepted with a certain probability close to 1. The temperature

cycle $k = \{0, 1, \ldots\}$ increases with time, for instance, with the number of fuzzing executions. The cooling schedule computes the *current temperature* T_k for cycle k. In our case, the current temperature determines the energy assigned to a seed. Intuitively, if the temperature is still *high*, a seed s_{10} that exercises a path with a high path distance is assigned the *same energy* as a seed s_0 that exercises a path with a low path distance. As the temperature *approaches zero*, s_0 is assigned *most energy* while s_{10} may not be fuzzed at all. The most commonly used cooling schedule is the *exponential multiplicative*:

$$T_{\text{exp}} = T_0 \cdot \alpha^k \tag{6.1}$$

where α is a constant smaller than the unit and typically $0.8 \leq \alpha \leq 0.99$. Figure 6-3 shows a plot of the exponentia multiplicative cooling schedule for an arbitrary value α and normalized initial temperature T_0.

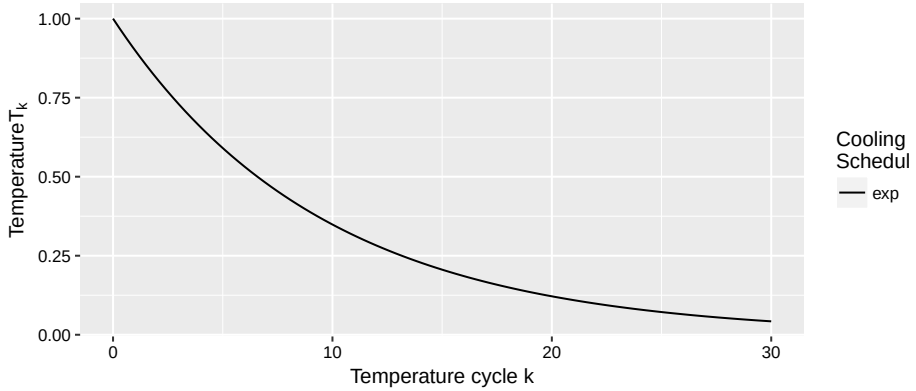


Figure 6-3: Rate of convergence for the exponential multiplicative cooling schedule, $T_k = 0.9^k$ where $T_0 = 1$.

6.3.2 Coverage-based Greybox Fuzzing

As explained in Section 2.4, *Coverage-based greybox fuzzing* (CGF) [22, 4] uses lightweight instrumentation to gain coverage information. For

instance, AFL's instrumentation captures basic block transitions, along with coarse branch-taken hit counts. CGF uses the coverage information to decide *which generated inputs to retain* for fuzzing, *which input to fuzz next* and *for how long.*

Algorithm 3 Coverage-based Greybox Fuzzing (adapted from [22])

Input: Seed Inputs S

```
 1: repeat
 2:     s = chooseNext(S)
 3:     p = assignEnergy(s)                    // Our Modifications
 4:     for i from 1 to p do
 5:         s' = mutate_input(s)
 6:         if t' crashes then
 7:             add s' to S✗
 8:         else if isInteresting(s') then
 9:             add s' to S
10: until timeout reached or abort-signal
```

Output: Crashing Inputs $S_{\text{✗}}$

Algorithm 3 shows an algorithmic sketch of how CGF works. The fuzzer is provided with a set of seed inputs S and chooses inputs s from S in a continuous loop until a timeout is reached or the fuzzing is aborted. The *selection* is implemented in chooseNext. For instance, AFL essentially chooses seeds from a circular queue in the order they are added. For the selected seed input s, the CGF determines the number p of inputs that are generated by fuzzing s as implemented in assignEnergy (line 3). This is also where the (temperature-based) power schedules are implemented. Then, the fuzzer generates p new inputs by randomly mutating s according to defined mutation operators as implemented in mutate_input (line 5). AFL uses bit flips, simple arithmetics, boundary values, and block deletion and insertion strategies to generate new inputs. If the generated input s' is covers a new branch, it is added to the circular queue (line 9). If the generated input s' crashes the program, it is added to the set $S_{\text{✗}}$ of crashing inputs (line 7). A crashing input that is also interesting is marked as *unique*

crash.

CGF as Markov Chain. Böhme et al. [22] showed that coverage-based greybox fuzzing can be modelled as a Markov chain. A *state* i is a specific path in the program. The *transition probability* p_{ij} from state i to state j is given by the probability that fuzzing the seed which exercises path i generates a seed which exercises path j. The authors found that a CGF exercises certain (high-frequency) paths significantly more often than others. The *density of the stationary distribution* formally describes the likelihood that a certain path is exercised by the fuzzer after a certain number of iterations. Böhme et al. developed a technique to gravitate the fuzzer towards low-frequency paths by adjusting the number of fuzz generated from a seed depending on the density of the neighborhood. The number of fuzz generated for a seed s is also called the *energy* of s. The energy of a seed s is controlled by a so-called *power schedule*. Note that energy is a property that is local to a state in the Markov chain unlike temperature which is global in simulated annealing.

Simulated annealing is a Markov Chain Monte Carlo approach. Since CGF can be modelled as Markov chain, it should be possible to employ such optimization techniques on top of CGF. In this work, we explore this possibility and develop a novel power schedule that integrates ideas from SA to direct the fuzzer towards a given set of targets.

6.4 Directed Greybox Fuzzing

Our main objective is the development of a lightweight, search-based vulnerability detection technique that works out-of-the-box for large-scale, file-processing programs and libraries. We pose three additional requirements: 1) It must be *easily parallelizable*, such that we

can assign computing power as and when needed. 2) It must allow to specify *multiple target locations*, like the set of changed statements in a commit or the set of critical system calls. 3) It must *not conduct any program analysis during runtime* so that all heavy-weight analysis should be conducted at compile-time. We lay the groundwork for the technique i) by defining measures of distance between program input and targets on an abstraction of the program (e.g., the control-flow or the call graph), and ii) by defining a power schedule that integrates the exponential muliplicative cooling schedule, which is the most commonly used schedule, from simulated annealing and the original power schedule of the AFL CGF.

6.4.1 A Measure of Distance Between the Exercised Path and Multiple Targets

Given a path ξ in a directed graph G where some nodes Γ are marked as *targets*, we define the distance $d(\xi, \Gamma)$ between path ξ and all targets Γ as follows. Let the *node distance* $d(n, n')$ be computed as the number of edges along the shortest path between nodes n and n' in the directed graph G. Let the *target distance* $d(n, \Gamma)$ between a node n and all targets Γ in G be computed as the harmonic mean of the logarithm of the node distance between n and any *reachable* target $\gamma \in \Gamma$:

$$d(n, \Gamma) = \begin{cases} 0 & \text{if } \Theta(n, \Gamma) = \emptyset \\ \left[\sum_{\gamma \in \Theta(n,\Gamma)} \log(\epsilon + d(n, \gamma))^{-1}\right]^{-1} & \text{otherwise} \end{cases} \tag{6.2}$$

where $\Theta(n, \Gamma)$ is the set of all targets that are reachable from n in G and $\epsilon > 1$ is a constant that simply prevents the case where the parameter of the logarithm or the divisor is zero (e.g. when $d(n, \gamma) = 0$ or when

(a) Arithmetic Mean (b) Harmonic Mean

Figure 6-4: Difference between node distance defined in terms of arithmetic mean versus harmonic mean. Node distance is shown in the white circles. The targets are marked in gray.

$\forall\gamma \in \Theta(n,\Gamma).d(n,\gamma) = 1)$. The *harmonic mean* allows to distinguish between a node that is closer to one target and further from another and a node that is equi-distant from both targets. The arithmetic mean would assign both nodes the same node distance. Figure 6-4 provides an example. The *logarithm* reduces the influence of very distant outlier targets on the harmonic mean.

Let the *path distance* $d(\xi,\Gamma)$ between path ξ and all targets Γ in G be the arithmetic mean of the target distance between any node $n \in \xi$ and Γ, but ignoring nodes that have no reachable targets.

$$d(\xi,\Gamma) = \frac{\sum_{n\in\xi} d(n,\Gamma)}{|\{n \mid n \in \xi \wedge \Theta(n,\Gamma) \neq \emptyset\}|} \tag{6.3}$$

Given the set of paths Λ exercised by the current set of seeds, we define the *normalized path distance* $\tilde{d}(\xi,\Gamma)$ as the difference between the path distance of ξ to Γ and the minimum path distance of any path $\xi' \in \Lambda$ exercised by the current set of seeds to Γ divided by the difference between the maximum and the minimum path distance of any path $\xi' \in \Lambda$ exercised by the current set of seeds to Γ. Note that the normalized path distance

$\tilde{d} \in [0, 1]$.[3]

$$\tilde{d}(\xi, \Gamma) = \frac{d(\xi, \Gamma) - \text{minD}}{\text{maxD} - \text{minD}} \tag{6.4}$$

where

$$\text{minD} = \min_{\xi' \in \Lambda} (d(\xi', \Gamma)) \tag{6.5}$$

$$\text{maxD} = max_{\xi' \in \Lambda} (d(\xi', \Gamma)) \tag{6.6}$$

6.4.2 Temperature-based Power Schedule

A *Temperature-based Power Schedule* (TPS) assigns energy to a seed s depending on the current temperature T_k of the simulated annealing process and the distance of the path ξ that is exercised by s to the set of targets Γ in the call graph G. In simple terms, a seed that exercises a path that is "closer" to the targets is assigned more energy than a seed that exercises a path "further away" from the targets, and this energy difference increases as the temperature decreases. First, we *normalize* all measures to the range $[0, 1]$. We set $T_0 = 1$ such that $T_k \in [0, 1]$ and recall that distance $\tilde{d} \in [0, 1]$.

Generic TPS. Given the current temperature T_k that is computed according to the exponential multiplicative cooling schedule, a graph G, targets Γ in G, and path ξ in G, we define the *generic temperature-based power schedule* to assign energy p as

$$p(T_k, \xi, \Gamma) = (1 - \tilde{d}(\xi, \Gamma)) \cdot (1 - T_k) + 0.5T_k \tag{6.7}$$

[3]It is worth noting that a definition of normalized path distance as the arithmetic mean of the "normalized" target distance (w.r.t. min. and max. target distance) – in our experiments – resulted in the probability density being centered around a value much less than 0.5 with significant positive kurtosis. This resulted in substantially reduced energy for *every* seed. The definition of normalized path distance in Equation (6.4) reduces the kurtosis and nicely spreads the distribution between zero and one.

The behavior of the generic TPS is illustrated in Figure 6-5 for three values of T_k and d. Notice that energy $p \in [0, 1]$. Moreover, for $T_k = 1$, the generic TPS assigns the same energy to a seed exercising a path with a high path distance as to one exercising a path with a low path distance. A path that exercises all targets (i.e., $\tilde{d} = 0$) is assigned more and more energy as the temperature decreases.

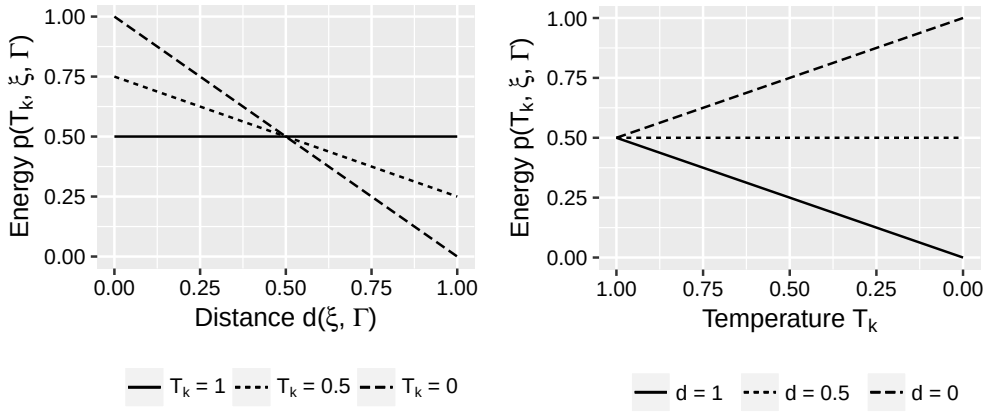


Figure 6-5: Impact of path distance $\tilde{d}(\xi, \Gamma)$ and temperature T_k on the energy $p(T_k, \xi, \Gamma)$ of the seed exercising path ξ.

In testing, we usually have only a limited time budget. Hence, we would like to specify a time t_x when the temperature-based power schedule should enter exploitation after sufficient time of exploration. We let the cooling schedule enter *exploitation* when $T_k \leq 0.05$. The adjustment of the generic TPS for values other than 0.05 is straightforward. Intuitively, at time t_x, the simulated annealing process is comparable to a classical gradient descent algorithm that "rejects" almost all seeds that are too far away from the targets. Given the *exponential cooling schedule* $T_{\text{exp}} = \alpha^k$, a time bound t_x when $T_{\text{exp}} = 0.05$, we compute the current temperature T_{exp}

at the current time t as follows

$$0.05 = \alpha^{k_x} \qquad \text{for } T_{\text{exp}} = 0.05;\ k = k_x \text{ in Eq. (6.1)} \tag{6.8}$$

$$k_x = \log(0.05)/\log(\alpha) \qquad \text{solving for } k_x \text{ in Eq. (6.8)} \tag{6.9}$$

$$T_{\text{exp}} = \alpha^{\frac{t}{t_x}\frac{\log(0.05)}{\log(\alpha)}} \qquad \text{for } k = \frac{t}{t_x}k_x \text{ in Eq. (6.1)} \tag{6.10}$$

$$= 20^{-\frac{t}{t_x}} \qquad \text{simplifying Eq. (6.10)} \tag{6.11}$$

Integrated TPS. AFL already has a power schedule to decide how many fuzz iterations it will conduct for a specific seed. This decision is made based on the execution time and input size of s, when s has been found, and how many ancestors s has. We would like to integrate AFL's pre-existing power schedule with our generic temperature-based power schedule and define the final integrated temperature-based power schedule. Let p_{afl} be the energy that AFL normally assigns to a seed. We compute the integrated TPS $\hat{p}$ as

$$\hat{p} = p_{\text{afl}} \cdot 2^{10\cdot(p(T_k,\xi,\Gamma)-5)} \tag{6.12}$$

where we call $f = 2^{10(p-0.5)}$ the temperature-based factor which controls the increase or reduction of energy assigned by AFL's power schedule.

Plots for the factor f for the Temperature-based Power Schedule (TPS) and two seeds with minimal and maximal path distance, respectively, are shown in Figure 6-6. When normalized path distance is minimal ($\tilde{d} = 0$), TPS approaches a factor of $2^5 = 32$ as the time t increases. When the normalized path distance is maximal ($\tilde{d} = 1$), TPS approaches a factor of $\frac{1}{2^5} = 1/32$.

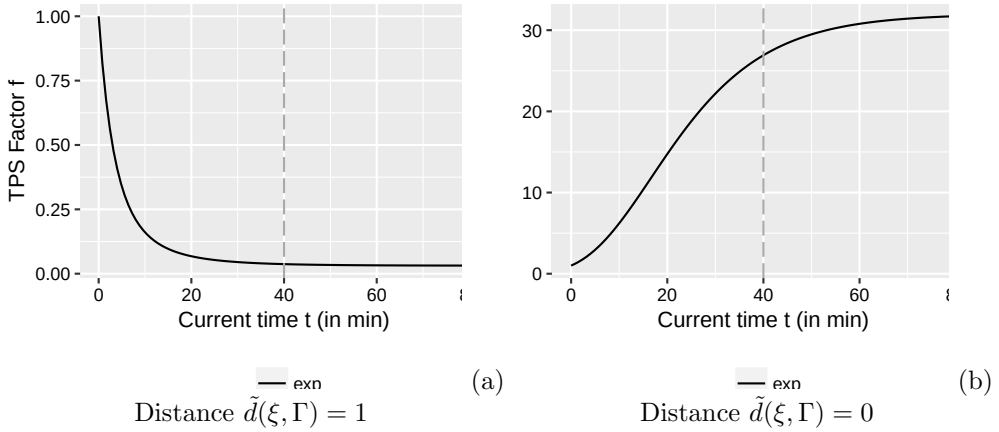


Figure 6-6: Temperature-based power factor which controls the energy that was originally assigned by AFL's power schedule ($t_x = 40$), (a) for seed with minimal path distance to all targets ($\tilde{d} = 0$) and (b) for a seed with maximal distance to all targets ($\tilde{d} = 1$). Notice the different scales on the y-axis.

6.5 Implementation

AFLGO implements directed coverage-based greybox fuzzing and takes program methods as targets. It is based on the AFL fuzzer (v2.35b). AFL represents the state-of-the-art of coverage-based greybox fuzzing, is behind hundreds of high-impact vulnerability discoveries [4], and has been shown to generate valid image files (JPEGs) from an initial seed that is virtually empty [71]. We modified the instrumentation at compile-time and the fuzzing component working at runtime, specifically ASSIGNENERGY in Algorithm 3.

6.5.1 All Program Analysis at Compile Time

During compile time, AFLGO extracts the call graph, marks the target functions, and computes the node distance values for all functions to the target functions. Given the names of the target methods, a python script computes the distances for all methods that can directly or indirectly call at least one target method. The names of the target methods can be

extracted automatically depending on the application, for instance, from the commit if the application is regression test generation. The *call graph* is constructed both, *statically and dynamically*. First, LLVM `opt` extracts the call graph using static analysis; `opt` is the LLVM optimizer which is capable of sophisticated program analysis during compile time. However, the extracted call graph may be incomplete, for instance, due to register-indirect jumps. Hence, the performance profiler `gprof` is used to track the method calls executed by the test suite which is provided in every subject. This information is then used to increase the completeness of the call graph. The nodes corresponding to the names of the target methods are marked in the call graph. Distance values are computed for each node in the call graph using the `graph-tool` package in python and according to the formula provided in Equation (6.2).

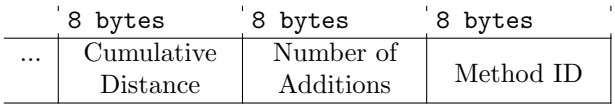


Figure 6-7: AFL shared memory – extended layout (x86-64)

In order to make AFLGO aware of distance to targets, we extended the standard assembler-based instrumentation of AFL in `afl-as`. The assembler reads the file containing the method names and the corresponding distance values. During instrumentation, the assembler knows which method it is currently instrumenting and passes the current method identifier and the corresponding distance value to the injected "trampoline".[4] The trampoline that is injected by AFLGO's assembler assumes that the shared memory that is passed by AFLGO during execution is extended by 24 bytes (Fig. 6-7). Let D be the set of distance values corresponding to each method that is executed by the seed. The

[4]The *trampoline* is a piece of code injected by the AFL assembler that is executed after each jump instruction to keep track of the covered control-flow edges.

first eight additional bytes are used to accumulate the cumulative node distance values (i.e., $\sum_{d \in D} d$) as and when the seed is executed. These are followed by eight bytes that contain the count of accumulated distance values (i.e., $|D|$). Thus, the first eight bytes allow us to compute the arithmetic mean of the distances of the exercised nodes as in Equation (6.3) (i.e., $\left(\sum_{d \in D} d\right) / |D|$). The last eight bytes contain the identifier for the current method. The trampoline is injected at each branching point in a function. Hence, we would accumulate more distance values for longer functions, unnecessarily biasing the search. We use the *current* method identifier in the trampoline and the *previous* method identifier in the shared memory, to accumulate distance values only when the current method actually changes.

6.5.2 Efficient Search at Runtime

In order to implement our the Temperature-based Power Schedule (TPS) into AFLGo, we extended the AFL coverage-based greybox fuzzer. For each execution of a generated test case (called fuzz), we pass an extended shared memory to the program under test (Fig. 6-7) and store the computed path distance (not yet normalized)[5] together with the fuzz if it is found interesting and added to the queue as new seed. We implemented the TPS shown in equation (6.11) by modifying the function `calculate_score` in `afl-fuzz`. This function normalizes the path distance and computes the time t since the fuzzer was started, before computing the temperature and required energy according to the exponential multiplicative cooling schedule. This function effectively implements the method assignEnergy in Algorithm 3 which decides how many fuzzing iterations should be executed for a seed. A more

[5]Recall that path distance is normalized w.r.t. the minimum and maximum path distance for the seeds currently in the queue (see Sec. 6.4.2).

common way to implement a meta-heuristic for search-based software testing is to base the decision of *which* seed to choose next on the distance of all seeds in the queue. However, we decided against modifying the method chooseNext in Algorithm 3 on empirical grounds. For most subjects in our experiments, the complete queue was fully processed in a matter of minutes which did not warrant the required computations needed for the re-ording of the queue.

We note that TPS can be implemented with only a few bit shifts, division, addition, and multiplication operations. In other words, the computation of the energy that is assigned to a seed is extremely efficient and there is *no program analysis at runtime*.

6.6 Experimental Evaluation

The *main objective* of our empirical investigations is to determine whether the directedness that is implemented into a coverage-based greybox fuzzer is effective in directing the search towards inputs that can reach the target functions. To this end, we conducted two main experiments to evaluate the effectiveness of AFLGo in reproducing crashes and patch testing to discover vulnerabilities. In crash reproduction experiment, AFLGo is guided by functions in crashing stack trace while in patch testing AFLGo is directed by the changes in a source code commit (a.k.a a patch). AFLGo would require users to choose a specific time-to-exploitation. So, we also investigate the sensitivity of our technique on the user-provided parameter and to identify a superior setting. The experiments help us to answer the following research questions.

RQ.1 Improvement of AFLgo over AFL. Is the extended power schedule effective in guiding the fuzzer towards a specified set of

targets? More specifically, does AFLGO reproduce crashes faster than AFL?

RQ.2 Sensitivity to Time-to-Exploitation Setting. How does the choice of time-to-exploitation t_x impact the efficiency of directed fuzzing? More specifically, does a particular choice of t_x make AFLGO generally faster in generating the crashing input than any other choice?

RQ.3 Patch testing for vulnerability detection. How does directed fuzzing perform when the objective is to reach the changed statements in a source code commit and expose program errors? More specifically, can AFLGO discover vulnerabilities in code patches?

6.6.1 Experimental Setup

6.1.1. Subjects

For the crash reproduction experiment, we selected a total of 18 vulnerabilities. We chose all eight (8) vulnerabilities in Binutils that were found by AFLFAST [22] and the Top-10 most recent vulnerabilities reported for LibPNG [97]. Binutils is a binary analysis tool and has almost one million Lines of Code (LoC) while LibPNG is an image library and has almost half a million LoC. Both are widely used open-source C projects. The vulnerabilities are identified by the CVE-ID and are discussed in more detail in the US National Vulnerability Database.

First, we needed to generate the Proof of Vulnerability (PoV) for each CVE using undirected AFL in order to collect a test case that actually produces the required stack trace. Notice that in practice an in-house

developer collects the stack trace from bug reports sent from users' machines. In order to check, whether a specific CVE has been exposed, we executed the crashing test case on the version where that CVE is patched. However, in several cases AFL was not able to generate a crashing input for a vulnerability in 20 runs of eight hours.[6] The remaining CVEs for which we could collect the stack trace are shown in Figure 6-8.

Program	**CVE-ID**	**Type of Vulnerability**
LibPNG [97]	CVE-2011-2501	Invalid Read
LibPNG [97]	CVE-2011-3328	Division by Zero
LibPNG [97]	CVE-2015-8540	Invalid Read
Binutils [22]	CVE-2016-4487	Invalid Write
Binutils [22]	CVE-2016-4488	Invalid Write
Binutils [22]	CVE-2016-4489	Invalid Write
Binutils [22]	CVE-2016-4490	Write Access Violation
Binutils [22]	CVE-2016-4491	Stack Corruption
Binutils [22]	CVE-2016-4492	Write Access Violation
Binutils [22]	CVE-2016-6131	Write Access Violation

Figure 6-8: Subjects for Crash Reproduction.

For the patch testing for vulnerability detection experiment, we used AFLGo to test 1600 most recent revisions of Binutils[7], from the one which incoporated the fixes for eight (8) vulnerabilities found by AFLFAST[8] [22] to the newest version on trunk. In this experiment, AFLGo was guided by code changes in each revision. We first ran a script to filter out all commits that have no code change. Afterwards, we ran one instance of AFLGO for each program in a Binutils revision - no shared queue was used. It is worth noting that in this experiment, we wanted to test not only how a program handles input file but also how it parses and processes input arguments. To this end, we used a tool named `afl-argv` developed by our

[6]Eight hours of fuzzing might not be enough for many vulnerabilities that are notoriously hard to discover. Running 20 instances of AFL for 8 hours, we could not generate a PoV for LibPNG CVEs: 2011-3026, 2011-3048, 2011-3464, 2012-3386, 2013-6954, 2014-0333, 2014-9495, 2015-8126, or for Binutils CVE 2016-2226.

[7]Git repository at git://sourceware.org/git/binutils-gdb.git

[8]Its commit hash is fa3fcee7b8c73070306ec358e730d1dfcac246bf

team. Essentially, `afl-argv` defines a simple file structure to keep data for both input arguments and input file(s). Once AFLGO produces a test case, `afl-argv` interpretes the test case and decomposes it into arguments and files based on the defined structure and send them to the program under test.

6.1.2. Settings

Almost all vulnerabilities in Binutils are exposed in less than eight hours [22]. So, we set a *timeout* for eight (8) hours and the default time to exploitation t_x to seven (7) hours. We ran a single instance of AFLGO along with a single instance of AFL (i.e. no shared queue) and measured the time to exposure (TTE). We repeated this experiment 20 times to gain statistical power.

The fuzzer instances for Binutils are seeded with an empty input. Thus, the fuzzer constructs the required binaries completely on its own. The fuzzer instances for LibPNG are seeded with all (4) valid PNG files from the corresponding AFL test suite.[9]

Recall that Time-To-Exploitation (t_x) is an independent variable that defines the time when the schedule should enter exploitation (i.e., $T_k = 0.05$). In the crash reproduction experiment, we investigate the impact of the choice of t_x on the efficiency of the technique. Specifically, we ran the experiment using the default value (seven (7) hours) and five (5) other values of t_x which are 1 minute, 10 minutes, 100 minutes, 1000 minutes and 1000 minutes.

[9]Test suite folder: `afl/testcases/images/png/*.png`.

6.1.3. Measures

In the crash reproduction experiment we used the following measures to evaluate the improvement of AFLGo over AFL.

Time-to-Exposure (*TTE*) is a dependent variable that measures the time taken from the start of the fuzzer until generating the first test case that exposes a given error. We determine which error a test case exposes by executing the failing test case on the set of fixed versions, where each version fixes just one error. If the test case passes on a fixed version, it is said to *witness* the corresponding error. If it is the first such test case, it is said to *expose* the error. We repeat each experimental setting 20 times and only report the mean time to exposure (μTTE) as the average over all measured TTE values for a specific setting.

Factor Improvement (*Factor*) is a measure of effect size and is defined as the μTTE of AFL divided by the μTTE of AFLGO for a given error. For instance, a *Factor* of 2 means that the average fuzzing campaign of AFL takes twice as long to expose a given error as the average fuzzing campaign of AFLGO. Values of $Factor > 1$ indicate that AFLGO outperforms AFL.

Vargha-Delaney statistic ($\hat{A}_{12}$) is a non-parametric measure of effect size [100]. It is also the recommended standard measure for the evaluation of randomized algorithms in software engineering [15]. Given a performance measure M (such as TTE) seen in m measures of X (such as AFLGO) and n measures of Y (such as AFL), the $\hat{A}_{12}$ statistic measures the probability that running algorithm X yields higher M values than running algorithm Y. We use the `VD.A` function from the `effsize` package in R to compute the $\hat{A}_{12}$ statistic. Values of $\hat{A}_{12} > 0.5$ indicate that AFLGO outperforms AFL.

6.1.4. Infrastructure

We executed all experiments on machines with an Intel Xeon CPU E5-2620v3 processor that has 24 logical cores running at 2.4GhZ with access to 64GB of main memory and Ubuntu 14.04 (64 bit) as operating system. We always utilized exactly 22 cores to keep the workload compareable and to retain two cores for other processes. Running our experiments on 20 machines with this equipment allowed us to run our experiments in four days that would normally take more than one year even on a recent PC with four logical cores.

6.6.2 Results and Analysis

CVE	Fuzzer	Successful runs	μTTE	Factor	$\hat{A}_{12}$
2011-2501 (LibPNG)	AFLGo	20	0h06m	2.81	**0.79**
	AFL	20	0h18m	–	–
2011-3328 (LibPNG)	AFLGo	20	0h40m	4.48	**0.94**
	AFL	18	3h00m	–	–
2015-8540 (LibPNG)	AFLGo	20	0m26s	10.66	**0.87**
	AFL	20	4m34s	–	–
2016-4487 (Binutils)	AFLGo	20	0h02m	1.64	0.59
	AFL	20	0h04m	–	–
2016-4488 (Binutils)	AFLGo	20	0h11m	1.53	**0.72**
	AFL	20	0h17m	–	–
2016-4489 (Binutils)	AFLGo	20	0h03m	2.25	**0.68**
	AFL	20	0h07m	–	–
2016-4490 (Binutils)	AFLGo	20	1m33s	0.64	**0.31**
	AFL	20	0m59s	–	–
2016-4491 (Binutils)	AFLGo	5	6h38m	0.85	0.44
	AFL	7	5h46m	–	–
2016-4492 (Binutils)	AFLGo	20	0h09m	1.92	**0.81**
	AFL	20	0h16m	–	–
2016-6131 (Binutils)	AFLGo	6	5h53m	1.24	**0.61**
	AFL	2	7h19m	–	–

Figure 6-9: Improvement of AFLGo over AFL in crash reproduction application. We run this experiment 20 times and highlight statistically significant values of $\hat{A}_{12}$ in bold. A run that does not reproduce the vuln. within 8 hours receives a TTE of 8 hours. CVEs 2016-4491 and 2016-6131 are difficult to find even in 24 hours [22].

RQ.1 Improvement of AFLgo over AFL

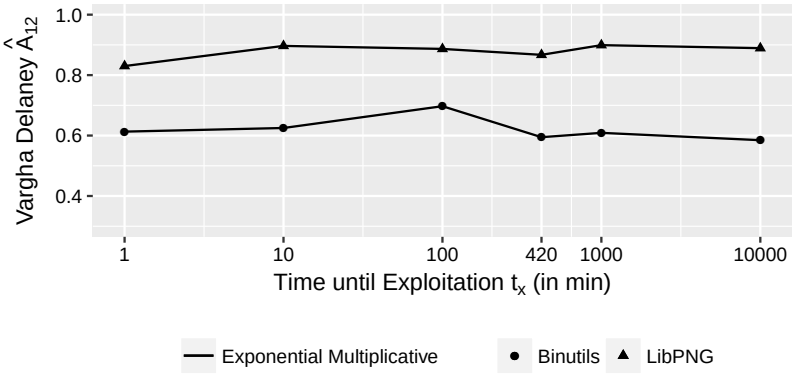
To reproduce the CVEs in LibPNG, AFLGO *is three (3) to 18 times faster than AFL.* More details are shown in Figure 6-9. For CVE-2015-8540, AFLGO needs only a few seconds to reproduce the vulnerability while AFL requires almost five minutes. For CVE-2011-3328, AFLGO spends merely half an hour while AFL requires three hours. For the remaining CVE (2011-2501), AFLGO can reproduce the crash in only six minutes while AFL takes more than three times as long.

To reproduce the CVEs in Binutils, AFLGO *is usually between 1.5 and 2 times faster than AFL.* There are two CVEs that are difficult to expose (2016-4491 and 2016-6131). In fact, both AFLGO and AFL took several hours in average to discover the CVEs. In case of CVE-2016-6131, AFLGO shows a clear improvement: AFLGO reproduces the crash for three times more runs and requires about one hour less time. AFL seems to exhibit better performance in only two CVEs (2016-4490 and 2016-4491). For CVE-2016-4490 it is exposed in a few seconds and at this scale the external impact is not negligible. For CVE-2016-4491, AFLGO and AFL are almost on par. However, it is worth noting that the results shown in Figure 6-9 are for a single time-to-exploitation setting (7 hours or 420 minutes). The sensitivity analysis in Figure 6-10 shows that there exist two settings (10 minutes and 100 minutes) which make AFLGO superior in case of CVE-2016-4491.

RQ.2 Sensitivity to Time-to-Exploitation Setting

On the average, AFLGO *is not particularly sensitive to the choice of time-to-exploitation t_x. However, for each vulnerability there seems to be an optimal value for time-to-exploitation.* In Figure 6-10-a, we show the time-to-exploitation t_x on a logarithmical scale. On the average, the

(a) Average effect size for LibPNG and Binutils (exp. and log. TPS)



(b) Effect size for each vulnerability

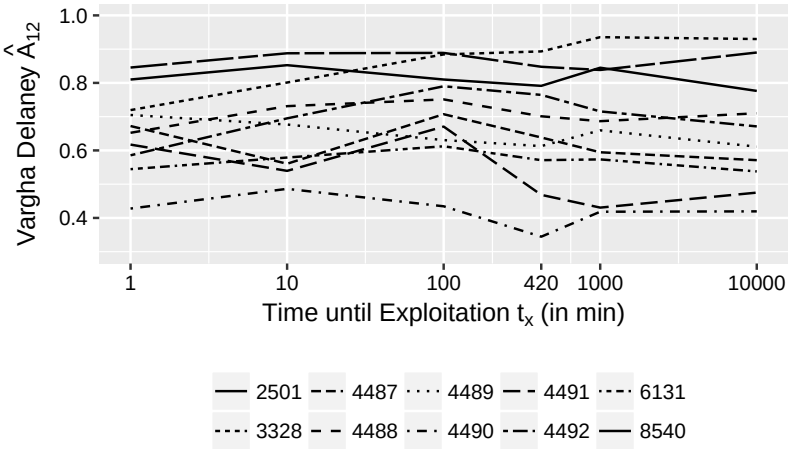


Figure 6-10: Sensitivity to the time to exploitation t_x. We show the individual improvement for each vulnerability.

improvement of AFLGO over AFL is not much different if $t_x = 1$ *minute* versus $t_x = 10000\,\text{min} \approx 1$ *week*. For Binutils, the best choice should be around $t_x = 100$ *minutes*. In fact, the improvement of AFLGO is 10 percentage points at $t_x = 100\,\text{min}$ than in our default experimental setting ($t_x = 420\,\text{min} = 7$ *hours*) that is discussed in RQ.1. However, Figure 6-10-b, depicts clearly that each vulnerability has a superior value for t_x. For the LibPNG vulnerability CVE-2011-3328, the optimal time-to-exploitation seems to be beyond 16 hours. For the Binutils vulnerability CVE-2016-4491, instead, the optimal time-to-exploitation seems to be between 1 and 2 hours. After the optimum the Vargha-Delaney measure drops by 20 percentage points. We investigated

whether hard-to-discover errors require a longer time-to-exploitation but found no obvious relationship.

Interpretation. Without further knowledge about the difficulty of reaching the targets, all choices t_x would be similarly effective. A priori, any choice is reasonable. However, there is clearly a sensitivity on these parameters for each error in particular. We believe that a suitable hyper-heuristic could adjust the choices of t_x during search process itself [56, 57]. Moreover, AFLGO allows to run several instances in parallel, where each instance can share the seeds found with the other instances via a *shared queue*. If sufficient computing resources are available, we suggest to run the several AFLGO instances with different choices for t_x to increase the chance that one is more efficient than the others.

RQ.3 Patch testing for Vulnerability Detection

In the motivating example, we have shown that AFLGO guided by the code changes in the commit rather than the stack trace (as used in crash reproduction application) successfuly exposes the famous Heartbleed vulnerability more than three times faster than AFL. In fact, AFLGO takes less than six (6) hours while AFL takes more than 20 hours. Moreover, AFLGO is much more deterministic than AFL; while AFLGO can expose Heartbleed in all of 30 runs, AFL succeeds in only 18 runs.

AFLGO not only exposes well-known vulnerablities like Heartbleed. By doing patch testing on 1600 revisions of Bintuils AFLGO discovers 14 zero-day vulnerabilities in three different utilities in Binutils; all these utilities (Readelf, Nm and Objdump) are widely used by security practitioners and software engineers to analyze program binaries. We have received (5) CVEs assigned to the most critical vulnerabilities among 14 discovered bugs. Moreover, several bugs are deeply hidden in

shared libraries (e.g. BFD and DWARF) which are part of several real-world applications such as Valgrind and GDB. Figure 6-11 lists all the 14 vulnerabilities[10] covering a variety types of bugs including heap-based buffer over-read, heap-based buffer over-write, use-after-free and NULL pointer dereference. While invalid write, use-after-free and NULL pointer dereference vulnerabilities are more likely to be exploitable, invalid read ones could lead to serious information leaks like Heartbleed or cause denial of service.

Program	**Bug-ID**	**CVE-ID**	**Type of Vulnerability**
Readelf	PR-21135	CVE-2017-7209	NULL pointer dereference
Readelf	PR-21137	CVE-2017-6965	Heap-based buffer over-write
Readelf	PR-21139	CVE-2017-6966	Use-after-free
Readelf	PR-21147	-	Heap-based buffer over-read
Readelf	PR-21148	-	Heap-based buffer over-read
Readelf	PR-21149	-	Heap-based buffer over-read
Nm	PR-21150	-	Heap-based buffer over-read
Objdump	PR-21151	-	Heap-based buffer over-read
Readelf	PR-21155	-	Memory Access violation
Readelf	PR-21156	CVE-2017-6969	Heap-based buffer over-read
Objdump	PR-21157	CVE-2017-7210	Heap-based buffer over-read
Objdump	PR-21158	-	Heap-based buffer over-read
Readelf	PR-21159	-	Heap-based buffer over-read
Objdump	-	-	Heap-based buffer over-read

Figure 6-11: Discovered zero-day vulnerabilities.

More interestingly, 12 out of 14 zero-day vulnerabilities can be attributed to the directedness of AFLGo. By manually investigating the code commits, the stack traces traversing by crashing inputs and the bug fixes written by Binutils's maintainers we found interesting statistics: three (3) out of 14 bugs are in target functions of AFLGo, nine (9) bugs are in so-called "critical" paths from entry functions (i.e. main functions of the test programs) to target functions. The remaining two (2) bugs are in functions which are invoked directly by some functions in the corresponding critical paths – their distances to critical paths are only

[10]The last bug in Objdump does not have Bug-ID because it is relevant to the critical `zlib` library so we sent email directly to Binutils' and zlib's maintainers to follow responsible disclosure policy.

one.

The experiment on patch testing also indicates that AFLGO can effectively detect bugs caused by incomplete bug fixes. Specifically, three bugs (PR-21155, PR-21156 and PR-21159) exist because Binutils' maintainers fixed incompletely our earlier reported bugs (PR-21137, PR-21156 and PR-21135 respectively. Notice that we reported two bugs in PR-21156, both the initial one and the one due to incomplete fix, so we got only one bug ID). Especially, for PR-21137 the maintainer had to submit three different bug fixes to fully resolve the problem.

The encouraging results in patch testing for vulnerability detection suggests that directed CGF technique (as implemented in AFLGO*) can be integrated into continuous integration/testing platforms like Jenkins [73] or Google OSS-Fuzz [74] to prevent bugs introduced by code changes in the evolution of software systems.*

6.7 Threats to Validity

The choice of subjects constitutes a threat to external validity. We select three real-world open-source programs (OpenSSL, LibPNG and Binutils) which are widely used and deployed. The choosen vulnerabilities represent a large variety of exploitable types of bugs. However, results may vary for different vulnerabilities, closed-source programs, programming languages, or architectures. We choose two applications of directed fuzzing to crash reproduction and regression test generation. Results may vary for other applications.

A common threat to internal validity for fuzzer experiments is the selection of initial seeds. However, since AFL and AFLGO are started with the same set of initial seeds (the empty seed or from the official AFL test suite), both fuzzers gain the same (dis-)advantage. Moreover,

AFLGO may not faithfully implement directed coverage-based greybox fuzzing as presented in this article, introducing a threat to construct validity. However, we make the source code and documentation of AFLGO available to the artifact evaluation committee and later to the general public for their scrutiny.

6.8 Chapter Summary

In this chapter, we have presented directed coverage based grey-box fuzzing (CGF) which allows us to direct the search to specific functions in the program, such as critical system calls, changed functions in a commit, or the methods in the stack trace of an unknown input. We demonstrate this directedness ability empirically by generating inputs which follow a given stack-trace, thereby also providing an efficient and effective solution for crash reproduction and by discovering zero-day vulnerabilities in patch testing which focuses on functions in code commits. The directedness is achieved by integrating an effective meta-heuristic search with the power schedule of a coverage-based greybox fuzzer where the power schedule decides how many inputs are generated from a seed that is a certain distance from the target functions.

Chapter 7

Bucketing Failing Tests via Symbolic Analysis

A common problem encountered while debugging programs is the overwhelming number of test cases generated by automated test generation tools like fuzzing, where many of the tests are likely to fail due to same bug. Some coarse-grained clustering techniques based on point of failure (PFB) and stack hash (CSB) have been proposed to address the problem. In this chapter, we present a new symbolic analysis-based clustering algorithm that uses the *semantic reason* behind failures to group failing tests into more "meaningful" clusters.

7.1 Introduction

Software debugging is a time consuming activity. Several studies [28], [39], [47], [51], [85] have proposed clustering techniques for failing tests and proven their effectiveness in large-scale real-world software products. The Windows Error Reporting System (WER) [47] and its improvements such as ReBucket [39] try to arrange error reports into various "buckets"

or clusters. WER employs a host of heuristics involving module names, function offset and other attributes. The Rebucket approach (proposed as an improvement to WER) uses specific attributes such as the call stack in an error report.

Although the techniques have been applied widely in industry, there are three common problems that they can suffer from (as mentioned in [47]). The first problem is *"over-condensing"* in which the failing tests caused by multiple bugs are placed into a single bucket. The second problem is *"second bucket"* in which failing tests caused by one bug are clustered into different buckets. The third one, *"long tail"* problem, happens if there are many small size buckets with just one or a few tests. For example, using failure type and location (as used in KLEE [28]) for clustering tests are more likely to suffer from both *over-condensing* and *second bucket* problems as they would group all tests that fail at the same location, completely insensitive to the branch sequence and the call-chain leading to the error. Call stack similarity for clustering tests also suffers from the "over-condensing" and "second bucket" problems because it is insensitive to the intraprocedural program paths (i.e. the conditional statements within functions). One of the main reasons why techniques in [28], [39], [47], [51], [85] suffer from these problems is that they do not take program *semantics* into account.

In this work, we propose a novel technique to cluster failing tests via symbolic analysis. Unlike previous work that drive bucketing directly from error reports, we adapt symbolic path exploration techniques (like KLEE [28]) to cluster (or bucket) the failing tests on-the-fly. We drive bucketing in a manner such that tests in each group fail due to the same *reason*. Since we use symbolic analysis for clustering, our technique leads to more accurate bucketing; that is (a) tests for two different bugs are less likely

to appear in the same bucket, and (b) tests showing the same bug are less likely to appear in different buckets. We experimentally evaluate our semantics-based bucketing technique on a set of 21 programs drawn from five repositories: IntroClass, Coreutils, SIR, BugBench and exploit-db. Our results demonstrate that our symbolic analysis based bucketing technique is effective at clustering tests: for instance, the **ptx** program (in our set of benchmarks) generated `3095` failing tests which were grouped into `3` clusters by our technique. Similarly, our tool clustered `4510` failing tests of the **paste** program into `3` clusters.

In addition to bucketing failures, our tool provides a *semantic characterization* of the reason of failure for the failures in each cluster. This characterization can assist the developers better understand the nature of the failures and, thus, guide their debugging efforts. The existing approaches are not capable of defining such an accurate charaterization of their clusters (other than saying that all tests fail at a certain location or with a certain stack configuration).

While our algorithm is capable of bucketing tests as they are generated via a symbolic execution engine, it is also capable of clustering failures in existing test-suites by a post-mortem analysis on the set of failures.

The contributions of this work are as follows:

- We propose an algorithm to efficiently cluster failing test cases, both for the tests generated automatically by symbolic execution as well as tests available in existing test-suites. Our algorithm is based on deriving a *culprit* for a failure by comparing the failing path to the nearest correct path. As we use semantic information from the program to drive our bucketing, we are also able to derive a *characterization* of the reason of failure of the tests grouped in a cluster. The existing approaches are not capable of defining such

characterization for the clusters they produce.

- We implement a prototype of the clustering approach on top of the symbolic execution engine KLEE [28]. Our experiments on 21 programs show that our approach is effective at producing more meaningful clusters as compared to existing solutions like the point of failure and stack hash based clustering.

7.2 Overview

We illustrate our technique using a motivating example in Listing 7.1. In the `main()` function, the code at line 27 manages to calculate the value of $(2^x + x! + \sum_{i=0}^{y} i)$ in which x and y are non-negative integers. It calls three functions, `power()`, `factorial()` and `sum()`, to calulate 2^x, $x!$ and sum of all integer numbers from 0 to y. While `sum()` is a correct implementation, both `power()` and `factorial()` are buggy.

In the `power()` function, the programmer attempts an optimization of saving a multiplication: she initializes the result (the integer variable `pow()`) to `2` (line 2) and skips the multiplication at line 5 if n equals `1`. However, the optimization does not handle the special case in which n is zero. When n is zero, the loop is not entered and the function returns `2`: it is a wrong value since 2^0 must be 1. Meanwhile, in the `factorial()` function the programmer uses a wrong condition for executing the loop at line 13. The correct condition should be $i \leq n$ instead of $i < n$. The incorrect loop condition causes the function to compute factorial of $n - 1$ so the output of the function will be wrong if $n \geq 2$.

We can use a symbolic execution engine (like KLEE) to generate test cases that expose the bugs. In order to do that, we first mark the variables x and y as symbolic (line 25) and add an assert statement at line 28. The

assertion is used to check whether the calculated value for $2^x + x! + \sum_{i=0}^{y}$ (as stored in *val*) is different from the expected value which is fetched from `golden_output()`.

The specification *oracle* `golden_output()` can be interpreted in many ways depending on the debugging task: for example, it can be the previous version of the implementation when debugging regression errors, or the expected result of each test when run over a test-suite. For the sake of simplicity, we add an `assume()` statements at line 26 to bound values of symbolic variables x and y.

```
1  unsigned int power(unsigned int n) {
2      unsigned int i, pow = 2;
3  /* Missing code: if (n == 0) return 1; */
4      for(i=1; i<=n; i++) {
5          if(i==1) continue;
6          pow = 2*pow;
7      }
8      return pow;
9  }
10 unsigned int factorial(unsigned int n) {
11     unsigned int i,result = 1;
12 /* Incorrect operator: < should be <= */
13     for(i=1;i<n;i++)
14         result = result*i;
15     return result;
16 }
17 unsigned int sum(unsigned int n) {
18     unsigned int result = 0, i;
19     for (i=0; i<=n; i++)
20        result += i;
21     return result;
22 }
23 int main() {
24     unsigned int x, y, val, val_golden;
25     make_symbolic(x, y);
26     assume(x<=2 && y<=2);
27     val = power(x)+factorial(x)+sum(y);
28     assert(val == golden_output(x, y));
29     return 0;
30 }
```

Listing 7.1: Motivating example

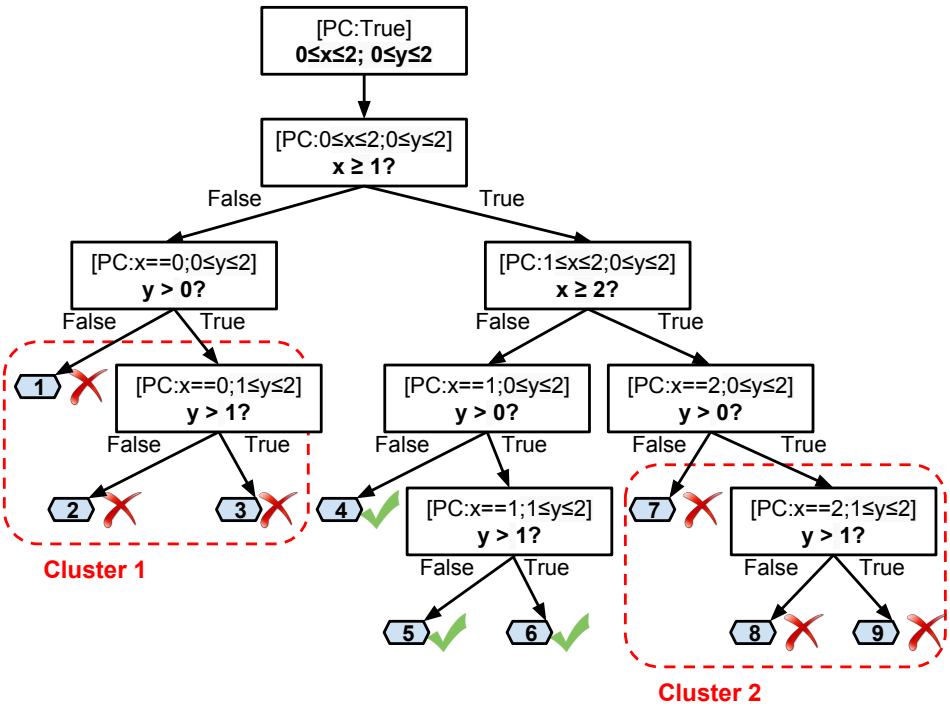


Figure 7-1: Symbolic execution tree for motivating example

Figure 7-1 shows the symbolic execution tree that KLEE would explore when provided with this example. In this work, we use the term **failing path** to indicate program paths that terminate in error. The error can be assertion violation or run-time error detected by symbolic execution engine such as divide-by-zero or memory access violation (as supported in KLEE). In contrast, the term **passing path** indicates paths that successfully reach the end of the program (or the *return* statement in the intraprocedural setting) with no errors.

As shown in Figure 7-1, KLEE explores 9 feasible executions and detects 6 failing paths; the paths are labeled from 1 to 9 in the order tests are generated while following the Depth-First-Search (DFS) search strategy. If we apply failure location based or call-stack based bucketing techniques, *both of them will place all 6 failing tests in a single cluster* as there is only one failure location at line 28, and the call stacks are identical when the failure is triggered. Hence, both the techniques suffer from the "over-

condensing" problem as the failures are due to two different bugs (in the `power()` and `factorial()` functions).

Let us now present our approach informally: given a failing test t encountered during symbolic exploration, our algorithm compares the path condition of t with the path condition of a successful test t' that has the *longest common prefix* with t. The branch b at which the execution of t and t' differ is identified as the *culprit branch* and the branch condition at b which leads to the failing path is identified as the *culprit constraint*—the "reason" behind the failure of t. Intuitively, the reason behind blaming this branch for the failure is that the failing path t could have run into the passing execution t'—only if this branch b had not misbehaved!

Table 7.1: Clustering result: Symbolic analysis

Path ID	Test Case	Path Condition	Culprit Constraint	Clus. ID
1	x=0, y=0	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x < 1)} \wedge (y \leq 0)$	$(x < 1)$	1
2	x=0, y=1	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x < 1)} \wedge (y > 0) \wedge (y \leq 1)$	$(x < 1)$	1
3	x=0, y=2	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x < 1)} \wedge (y > 0) \wedge (y > 1) \wedge (y \leq 2)$	$(x < 1)$	1
4	x=1, y=0	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x \geq 1) \wedge (x < 2)} \wedge (y \leq 0)$	NA	NA
5	x=1, y=1	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x \geq 1) \wedge (x < 2)} \wedge (y > 0) \wedge (y \leq 1)$	NA	NA
6	x=1, y=2	$(0 \leq x, y \leq 2) \wedge \boldsymbol{(x \geq 1) \wedge (x < 2)} \wedge (y > 0) \wedge (y > 1) \wedge (y \leq 2)$	NA	NA
7	x=2, y=0	$(0 \leq x, y \leq 2) \wedge (x \geq 1) \wedge \boldsymbol{(x \geq 2)} \wedge (y \leq 0)$	$(x \geq 2)$	2
8	x=2, y=1	$(0 \leq x, y \leq 2) \wedge (x \geq 1) \wedge \boldsymbol{(x \geq 2)} \wedge (y > 0) \wedge (y \leq 1)$	$(x \geq 2)$	2
9	x=2, y=2	$(0 \leq x, y \leq 2) \wedge (x \geq 1) \wedge \boldsymbol{(x \geq 2)} \wedge (y > 0) \wedge (y > 1) \wedge (y \leq 2)$	$(x \geq 2)$	2

Table 7.1 presents the result produced by our clustering algorithm (refer to Figure 7-1 for the symbolic execution tree). The failing tests 1-3 fail due to the bug in the `power()` function. The *culprit constraint* or "reason" for these failures is attributed as $x < 1$, since it is the condition on the branch where these failing tests diverge from their nearest passing test (Test 4), after sharing the *longest common prefix* $((0 \leq x \leq 2) \wedge (0 \leq y \leq 2))$. Hence, we create the first cluster (Cluster 1) and place tests 1-3 in it, with the characterization of the cluster as $(x < 1)$. Similarly, the failing tests 7-9 (failing due to the bug in `factorial()`) share the longest common

prefix $((0 \leq x \leq 2) \wedge (0 \leq y \leq 2) \wedge (x \geq 1))$ with Test 4; thus, the culprit constraint for tests 7-9 is inferred as $(x \geq 2)$. Hence, these tests are placed in Cluster 2 with the *characterization* $(x \geq 2)$. Note that the culprit constraints $(x < 1)$ and $(x \geq 2)$ form a neat *semantic characterization* of the failures in these two clusters.

Summary. In this example, our semantic-based bucketing approach correctly places 6 failing tests into 2 different clusters. Unlike the two compared techniques, it does not suffer from the "over-condensing" problem, and therefore, yields a more *meaningful* clustering of failures. Moreover, we provide a *semantic characterization* for each cluster that can assist developers in their debugging efforts. In fact, the characterization for `Cluster1` $(x < 1)$ exactly points out the bug in `power()` (as x is non-negative integer, $x < 1$ essentially implies that x equals zero). Likewise, the characterization for `Cluster2` $(x \geq 2)$ hints the developer to the wrong loop condition in the `factorial()` function (as the loop is only entered for $x \geq 2$). We, however, emphasize that our primary objective is **not** to provide root-causes for bugs, but rather to enable a good bucketing of failures.

7.3 Reasons of Failure

The *path condition* ψ_p of a program path p is a logical formula that captures the set of inputs that exercise the path p; i.e. ψ_p is true for a test input t if and only if t exercises p. We say that a path p is *feasible* if its path condition ψ_p is satisfiable; otherwise p is *infeasible*. We record the *path condition* ψ_p for a path p as a list of conjuncts l_p. Hence, the *size* of a path condition $(|\psi_p|)$ is simply the cardinality of the list l_p. We also assume that as symbolic execution progresses, the branch constraints

(encountered during the symbolic execution) are recorded in the path condition *in order*. This enables us to define *prefix(i, ψ_p)* as the prefix of length i of the list l_p that represents the path condition ψ_p. Hence, when we say that two paths p and q have a *common prefix* of length i, it means that *prefix(i, ψ_p)* = *prefix(i, ψ_q)*.

Inference of *suitable reasons* behind failures is central to clustering of failing tests: if the reasons inferred are too strong (*i.e.*, they are not general enough), tests that fail due to the same reason may form different clusters; we refer to the same as *under-clustering* or the *second bucket* problem. On the other hand, if constraints in the reason are too weak, then test cases that correspond to different reasons of failure may get clustered together; we refer this as *over-clustering* or *over-condensing* problem. We attribute the "reason of failure" of a failing path to a branch condition along the failing path such that there exists a passing path sharing the longest possible prefix with the failing path.

Definition 1 (Culprit Constraint)**.** *Given a failing path π_f with a path condition ψ_f (as a conjunct $b_1 \wedge b_2 \wedge \cdots \wedge b_i \wedge \ldots b_n$) and an exhaustive set of all feasible passing paths Π, we attribute b_i (the i-th constraint where i ranges from 1 to n) as the culprit constraint if and only if $i-1$ is the maximum value of j $(0 \leq j < n)$ such that $prefix(j, \psi_f) = prefix(j, \psi_p)$ among all passing paths $p \in \Pi$.*

We use the *culprit constraint* (as a symbolic expression) as the reason why the error path "missed" out on following the passing path; in other words, the failing path could have run into a passing path, only if the branch corresponding to the culprit constraint had not *misbehaved*. Our heuristic of choosing the culprit constraint in the manner described above is primarily designed to achieve the following objectives:

- **Minimum change to Symbolic Execution Tree**: Our technique targets well-tested production-quality programs that are "almost" correct; so, our heuristic of choosing the latest possible branch as the "culprit" essentially tries to capture the intuition that the symbolic execution tree of the *correct* program must be similar to the symbolic execution tree of the faulty program. *Choosing the latest such branch as the culprit is a greedy attempt at encouraging the developer to find a fix that makes the minimum change to the current symbolic execution tree of the program.*
- **Handle "burst" faults**: In Figure 7-1, all paths on one side of the node with $[PC : 1 \leq x \leq 2; 0 \leq y \leq 2]$ fail. So, the branching predicate for this node, $x \geq 2$, looks "suspicious". Our heuristic of identifying the latest branch as the culprit is directed at handling such scenarios of "burst" failures on one side of a branch.

7.4 Clustering Framework

7.4.1 Clustering Algorithm

Algorithm 4 shows the core steps in dynamic symbolic execution with additional statements (highlighted in grey) for driving test clustering. The algorithm operates on a representative imperative language with assignments, assertions and conditional jumps (adapted from [16], [66]). A symbolic executor maintains a state (l, pc, s) where l is the address of the current instruction, pc is the path condition, and s is a symbolic store that maps each variable to either a concrete value or an expression over input variables. At line 3, the algorithm initializes the worklist with an initial state pointing to the start of the program $(l_0, true, \emptyset)$: the first instruction is at l_0, the path condition is initialized as $true$ and the initial

Algorithm 4 Symbolic Exploration with Test Clustering

```
1:  procedure SymbolicExploration(l_0, W)
2:      C ← {}; passList ← [ ]; failList ← [ ]      ▷ initialization for bucketing
3:      W ← {(l_0, true, ∅)}                          ▷ initial worklist
4:      while W ≠ ∅ do
5:          (l, pc, s) ← pickNext(W)
6:          S ← ∅
7:          switch instrAt(l) do                      ▷ execute instruction
8:              case v := e                           ▷ assignment instruction
9:                  S ← {(succ(l), pc, s[v → eval(s, e)])}
10:             case if (e) goto l'                   ▷ branch instruction
11:                 e ← eval(s, e)
12:                 if (isSat(pc ∧ e) ∧ isSat(pc ∧ ¬e)) then
13:                     S ← {(l', pc ∧ e, s), (succ(l), pc ∧ ¬e, s)}
14:                 else if (isSat(pc ∧ e) then
15:                     S ← {(l', pc ∧ e, s)}
16:                 else
17:                     S ← {(succ(l), pc ∧ ¬e, s)}
18:             case assert(e)                        ▷ assertion
19:                 e ← eval(s, e)
20:                 if (isSat(pc ∧ ¬e)) then
21:                     testID ← GenerateTest(l, pc ∧ ¬e, s)
22:                     pc' ← ConvertPC(pc ∧ ¬e)
23:                     AddToList(failList,(testID,pc'))
24:                     continue
25:                 else
26:                     S ← {(succ(l), pc ∧ e, s)}
27:             case halt                             ▷ end of path
28:                 testID ← GenerateTest(l,pc,s)
29:                 pc' ← ConvertPC(pc)
30:                 AddToList(passList,(testID,pc'))
31:                 if failList ≠ [ ] then
32:                     ClusterTests(C,passList,failList)
33:                     failList ← [ ]                ▷ empty failing list
34:                 continue
35:         W ← W ∪ S                                 ▷ update worklist
36:     if failList ≠ [ ] then
37:         ClusterTests(C,passList,failList)
38: end procedure
```

store map is empty.

The symbolic execution runs in a loop until the worklist W becomes empty. In each iteration, based on a search heuristic, a state is picked for execution (line 7). Note that to support failing test bucketing, the search strategy must be DFS or an instance of our clustering-aware strategy

(*clustering-aware search strategy* discussed in Section 7.4.2). A worklist S (initialized as empty) keeps all the states created/forked during symbolic exploration.

Algorithm 5 Clustering failing tests

```
 1: procedure ClusterTests(Clusters,passList,failList)
 2:     for (failID, failPC) ∈ failList do
 3:         maxPrefixLength ← 0
 4:         for (passID, passPC) ∈ passList do
 5:             curPrefixLength ← LCP(failPC, passPC)
 6:             if curPrefixLength > maxPrefixLength then
 7:                 maxPrefixLength ← curPrefixLength
 8:         reason ← failPC[maxPrefixLength+1]
 9:         Update(Clusters,failID,reason)
10: end procedure
11: ------------------------------------------------------
12: procedure Update(Clusters,failID,reason)
13:     for r ∈ Clusters.Reasons do
14:         if isValid(reason ⇒ r) then
15:             Clusters[r].Add(failID)
16:             return
17:         else if isValid(r ⇒ reason) then
18:             UpdateReason(Clusters[r], reason)
19:             Clusters[reason].Add(failID)
20:             return
21:     AddCluster(Clusters, reason, failID)
22: end procedure
```

If the current instruction is an assignment instruction, the symbolic store s is updated and a new state pointing to the next instruction is inserted into S (lines $8-9$). A conditional branch instruction is processed (line 10) via a constraint solver that checks the satisfiability of the branch condition; if both its branches are satisfiable, two new states are created and inserted into S. If only one of the branches is satisfiable, the respective state is added to S. For *assert* instructions, the symbolic execution checks the assert condition, and if it holds, a new program state is created and the state is added to S. If the condition does not hold, it triggers an assertion failure, thereby, generating a failing test case (we call the respective test case a "**failing test**"). Some symbolic execution engines (like KLEE [28]) perform run-time checks to detect failures like divide-by-zero and memory access violations; in this algorithm, the *assert* instruction is used to represent the

failures detected by such checks as well. On encountering a halt instruction, the symbolic execution engine generates a test-case for the path (we refer to such a test case as a "**passing test**"). The *halt* instruction represents a normal termination of the program.

To support clustering of tests, we define two new variables, $passList$ and $failList$, to store information about all explored passing and failing tests (respectively). For each test, we keep a pair ($testID$, $pathCondition$), where $testID$ is the identifier of the test generated by symbolic execution, and $pathCondition$ is a list of branch conditions (explained in Section 7.3). We also introduce a variable C that keeps track of all clusters generated so far; C is a map from a culprit constraint (cluster reason) to a list of identifiers of failing tests. The bucketing functionality operates in two phases:

Phase 1: Searching for failing and passing tests. The selected search strategy guides the symbolic execution engine through several program paths, generating test cases when a path is terminated. We handle the cases where tests are generated, and update the respective list ($passList$ or $failList$) accordingly. In particular, when a failing test case is generated, the path condition (pc) is converted to a list of branch conditions (pc'). The pair comprising of the list pc' and the identifier of the failing test case form a representation of the failing path; the pair is recorded in $failList$ (lines 23–24). The $passList$ is handled in a similar manner (lines 31–32).

Phase 2: Clustering discovered failing tests. Once a passing test is found (lines 35–37) or the symbolic execution engine completes its exploration (lines 42–43), the clustering function $ClusterTests$ will be invoked. The procedure $ClusterTests$ (Algorithm 5) takes three arguments: 1) all clusters generated so far ($Clusters$), 2) all explored passing tests ($passList$) and 3) all failing tests that have not been

clustered ($failList$). In this function, the culprit constraints of all failing tests in $failList$ is computed (lines 2–9) and, then, the function *Update* is called (line 10) to cluster the failing tests accordingly.

The *Update* function (Algorithm 5) can place a failing test into an existing cluster or create a new one depending on the culprit constraint (reason) of the test. We base our clustering heuristic on the intuition that the reason of failure of each test within a cluster should be *subsumed* by a core reason (r_c) represented by the cluster. Hence, for a given failing test f (with a reason of failure r_f) being clustered and a set of all clusters *Clusters*, the following three cases can arise:

- **There exists $c \in C$ such that r_c subsumes r_f**: in this case, we add the test f to the cluster c (line 18);
- **There exists $c \in C$ such that r_f subsumes r_c**: in this case, we *generalize* the core reason for cluster c by resetting r_f as the general reason for failure for tests in cluster c (lines 21–22);
- **No cluster reason subsumes r_f, and r_f subsumes no cluster reason**: in this case, we *create* a new cluster *c'* with the sole failing test f and attribute r_f as the core reason of failure for tests in this cluster (line 26).

7.4.2 Clustering-aware Search Strategy

It is easy to see that Algorithm 4 will yield the correct *culprit constraints* if the search strategy followed is DFS: once a failing path f_i is encountered, the passing path that shares the maximum common prefix with f_i is either the last *passing* path encountered before the failure, or is the next *passing* path after f_i (i.e. ignoring all failures in the interim). Hence, a depth-first traversal of the symbolic execution tree will always find the culprit constraints by constructing the *largest common prefix* of the failing paths

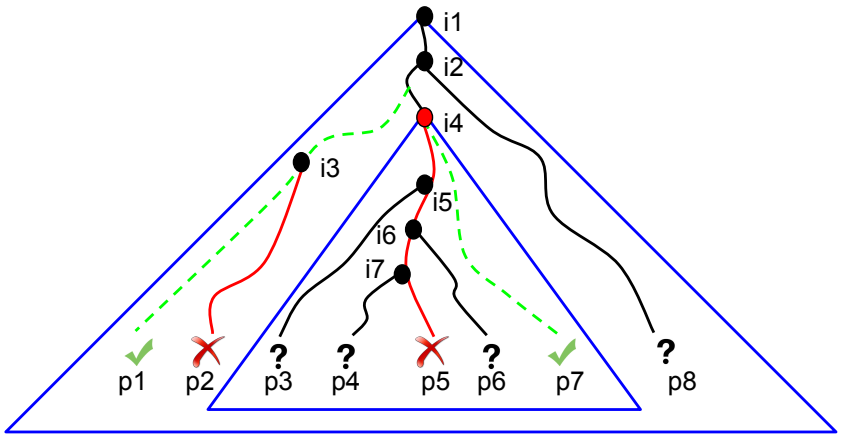


Figure 7-2: A Branching Tree

with at most two passing paths (the passing executions just before and just after encountering the failures).

However, DFS has a very poor coverage when used with a time-budget. Hence, we require search strategies different than DFS (like the Random and CoverNewCode strategies in KLEE) to achieve a good coverage. In fact, during our experiments, we could not trigger most of the failures in our benchmarks with DFS within reasonable timeouts.

We design a new *clustering-aware search strategy* (CLS) that is capable of discovering the precise culprit constraint while achieving a high coverage at the same time. CLS is built on a crucial observation that we *only* require DFS on a failing test to guide the search to its nearest passing test; on a passing test, the next test can be generated as per any search heuristic. Hence, one can implement any combination of suitable search strategies (to achieve high code coverage) while maintaining the use of DFS on encountering a failure (to identify the culprit constraint precisely).

We leverage a so-called *branching tree*, a data structure maintained by many symbolic execution engines (like KLEE) to record the symbolic execution tree traversed in terms of the branching/forking history (KLEE refers to it as the *process tree*). Let us illustrate as to how we combine

an arbitrary search strategy (*SS*) with DFS exploration to implement an instance of CLS using the branching tree in Figure 7-2. In the tree, $i1$–$i7$ are internal nodes while $p1$–$p8$ are leaf nodes. Note that in the following paragraphs, we will use the term (leaf) node and path interchangeably. Basically, CLS works in two phases:

Phase 1: *SS* searches for a failing test. The search heuristic *SS* searches for a failure using its own algorithm. Suppose *SS* first detects a failing path $p5$, it returns control to CLS that now switches to the DFS heuristic (to locate the "nearest" passing test, i.e. the one that has the longest common prefix with $p5$).

Phase 2: DFS looks for "nearest" passing test. Continuing with our example (Figure 7-2), by the time *SS* detects the failing path $p5$, assume that we have explored three paths $p1$, $p2$, $p7$ and successfully put the failing path $p2$ into its correct cluster. So, now only four active paths remain: $p3$, $p4$, $p6$ and $p8$. At this point, our CLS seach strategy uses another crucial observation: since $p7$ is a passing path and $i4$ is the closest common ancestor node of $p5$ and $p7$, the nearest passing path for $p5$ must be $p7$ or another passing path spawned from intermediate nodes $i5$, $i6$ or $i7$. Hence, we can reduce the search space for finding the nearest passing path for $p5$ from the space represented by outer blue triangle to the inner (smaller) triangle (as $p7$ is a passing path, it must be the nearest passing path if no "nearer" passing path is discovered in the subtree rooted at $i4$). We omit the details of how it is acheived for want of space.

If the symbolic execution is run with a timeout setting, the timeout can potentially fire while CLS is searching for the nearest passing path to a failing execution. In this case, we simply pick the nearest passing path to the failing execution among the paths explored so far to compute the culprit constraint.

Our technique is potent enough to cluster an existing test-suite by running the symbolic execution engine needs to run in a mode that the exploration of a path that is controlled by the failing test (like the "seed" mode in KLEE [29]). During path exploration, the first passing test encountered in a depth-first traversal seeded from the failing test t would necessarily be the passing test that has the longest common prefix with t. Thus, we can compute the *culprit constraint* accordingly, and use it to form a new cluster or update an existing cluster.

7.4.3 Generalize Reasons for Failure

Consider Listing 7.2: the program checks if the absolute value of each element in the array is greater than 0. The buggy assertion contains $>$ comparison instead of $\geq$), which would cause 10 failing test cases $\forall\ i \in \{0..9\}$. Since each array element is modeled as a different symbolic variable, all 10 cases are clustered separately.

```
1 int main() {
2   int arr[10], int i;
3   make_input(arr, sizeof(arr));
4   for (i = 0; i < 10; i++) {
5     if (a[i] < 0) a[i] = -a[i];
6     assert(a[i] > 0); // a[i] >= 0
7   }
8 }
```

Listing 7.2: Generalization for arrays

In such cases, we need to *generalize* errors that occur on different indices but due to the same core reason. For example, if the reason is: $arr[4] > 0 \wedge arr[4] < 10$, we change this formula to $\exists x\ (arr[x] > 0 \wedge arr[x] < 10)$. Note that this is only a heuristic, and our implementation allows the user to disable this feature.

7.5 Experimental Evaluation

We evaluated our algorithm on a set of 21 programs: three programs from *IntroClass* [52] (a micro benchmark for program repair tools) and the remaining eighteen real-world programs taken from four benchmarks-suites: eleven programs from Coreutils[1] version 6.10, three from SIR[41], one from BugBench[78] and three from exploit-db[2]. The three subject programs from exploit-db (downloaded them from the project's website) were used in [59]. The bugs in IntroClass, Coreutils, exploit-db and BugBench programs are real bugs, whereas the ones in SIR are seeded.

We manually inserted `assert` statements in the programs taken from the IntroClass benchmark to specify the test oracle, while all remaing 18 real-world programs were kept unchanged. During symbolic execution, the failing test cases are generated due to the violation of embedded assertions or triggering of run-time errors (captured by KLEE) like divide-by-zero and invalid memory accesses.

We compared our symbolic-analysis based (SAB) test clustering method to two baseline techniques: call-stack based (CSB) and point-of-failure based (PFB) clustering. While SAB refers to the implementation of our algorithm within KLEE, we implemented CSB and PFB on top of KLEE to evaluate our implementation against these techniques. Specifically, our implementation first post-processes the information of test cases generated by KLEE to compute the stack hash (on function call stack) and extract failure locations. Based on the computed and extracted data, they cluster the failing tests.

We conducted all of the experiments on a virtual machine created on a host computer with a 3.6 GHz Intel Core i7-4790 CPU and 16 GB of

Table 7.2: Test Clustering: number of clusters

Program	Repository	Size(kLOC)	#Fail. Tests	#Clus.(PFB)	#Clus.(CSB)	#Clus.(SAB)
median	IntroClass	1	7	1	1	5
smallest	IntroClass	1	13	1	1	3
syllables	IntroClass	1	870	1	1	5
mkfifo	Coreutils	38	2	1	1	1
mkdir	Coreutils	40	2	1	1	1
mknod	Coreutils	39	2	1	1	1
md5sum	Coreutils	43	48	1	1	1
pr	Coreutils	54	6	2	2	4
ptx	Coreutils	62	3095	16	1	3
seq	Coreutils	39	72	1	1	18
paste	Coreutils	38	4510	10	1	3
touch	Coreutils	18	406	2	3	14
du	Coreutils	41	100	2	2	8
cut	Coreutils	43	5	1	1	1
grep	SIR	61	7122	1	1	11
gzip	SIR	44	265	1	1	1
sed	SIR	57	31	1	1	1
polymorph	BugBench	25	67	1	1	2
xmail	Exploit-db	30	129	1	1	1
exim	Exploit-db	253	16	1	1	6
gpg	Exploit-db	218	2	1	1	1

RAM. The virtual machine was allocated 4 GB of RAM and its OS is Ubuntu 12.04 32-bit. For our experiments, we use the clustering-aware search strategy (CLS), enable array generalization and use a timeout of one hour for each subject program. KLEE is run with the `--emit-all-errors` flag to enumerate all failures.

7.5.1 Results and Analysis

Table 7.2 shows the results from our experiments on selected programs. `Size` provides the size of the program in terms of the number of LLVM bytecode instructions. `#Fail Tests` provides the number of failing tests. The rest of the columns provide the number of clusters (`#C`) for Point-of-failure (`PFB`), Stack Hash (`CSB`) and our Symbolic Analysis (`SAB`) based

Table 7.3: Test clustering: overhead

Program	#Pass. paths	#Fail. paths	Time (sec)	Overhead (%)
median	4	7	5	∼0
smallest	9	13	5	∼0
syllables	71	870	1800	4.35
mkfifo	291	2	3600	∼0
mkdir	326	2	3600	∼0
mknod	72	2	3600	∼0
md5sum	62449	48	3600	0.42
pr	540	6	3600	∼0
ptx	9	3095	3600	2.04
seq	445	72	1800	0.73
paste	3501	4510	3600	16.17
touch	210	406	3600	0.84
du	44	100	3600	0.81
cut	38	5	3600	∼0
grep	169	7122	3600	34.13
gzip	5675	265	3600	0.7
sed	3	31	3600	0.03
polymorph	3	67	3600	14.36
xmail	1	129	3600	0.06
exim	178	16	3600	0.03
gpg	10	2	3600	∼0

methods. Note that `#C(PFB)` also records the number of failing locations. As KLEE symbolically executes the LLVM bitcode, we show the size of the program in terms of the total lines of the LLVM bitcode instructions.

In several programs (like **ptx**, **paste**, **grep**) SAB places thousands of failing tests into managable number of clusters. Compared to CSB, in 12 out of 21 subjects (∼57%), our method produces more fine-grained clustering results. Compared to PFB, our technique expands the number of clusters to get a more fine-grained set in 10/21 subjects. However, our method also collapses the clusters in case the program has failures that are likely to be caused by the same bug but the failures occur at several different locations (like **ptx** and **paste**).

RQ1. Does our technique produce more fine-grained clusters?

In the experiments, we manually debugged and checked the root causes of failures in all subject programs. Based on that, we confirm that our SAB

approach does effectively produce more fine-grained clusters. For instance, as shown in Figure 7.4, the buggy **smallest** program, which computes the smallest number among four integer values, does not adequately handle the case in which at least two of the smallest integer variables are equal. For example, if d equals b, none of the four conditional statements (at lines 7, 9, 11 and 13) take the true branch; the result is incorrect as the variable `smallest` then takes an arbitrary value.

As shown in Listing 7.4, we instrumented the program to make it work with KLEE. During path exploration, KLEE generated 13 failing tests for this program and the CSB technique placed all of them into one cluster as they share the same call stack. However, our SAB approach created three clusters with the following reasons: (Cluster 1) $d \geq b$, (Cluster 2) $d \geq c$ and (Cluster 3) $d \geq a$. The reasons indeed show the corner cases that can trigger the bugs in the program. We observed similar cases in **median** and **syllables** programs (see Table 7.2).

```
1  case 'e':
2  if (optarg)
3    getoptarg (optarg, 'e', ...);
4  //...
5  break;
6  //other cases
7  case 'i':
8  if (optarg)
9    getoptarg (optarg, 'i', ...);
10 //...
11 break;
12 //other cases
13 case 'n':
14 if (optarg)
15   getoptarg (optarg, 'n', ...);
16 break;
```

Listing 7.3: Code snippet from 'pr'

In the subject program **pr** (a Coreutils utility), we found that 6 failing tests due to two different bugs are placed in two clusters on using stack

hash similarity. Meanwhile, our approach placed these 6 failing tests into 4 different clusters: one cluster contained 3 failing tests corresponding to one bug, and the other three clusters contain three failing tests of the second bug. Listing 7.3 shows a code snippet from **pr** that shows three call sites for the buggy function *getoptarg()* (at lines 3, 9 and 15). In this case, because all of the three call sites are in one function, so the stack hash based technique placed the three different failing paths in the same cluster. Similar cases exist in the **exim** and **du** applications.

```
1  int a, b, c, d, smallest;
2  make_symbolic(a, b, c, d);
3  assume(a>=-10 && a<=10);
4  assume(b>=-10 && b<=10);
5  assume(c>=-10 && c<=10);
6  assume(d>=-10 && d<=10);
7  if (a < b && a < c && a < d)
8      smallest = a;
9  if (b < a && b < c && b < d)
10     smallest = b;
11 if (c < b && c < a && c < d)
12     smallest = c;
13 if (d < b && d < c && d < a)
14     smallest = d;
15 assert(smallest == golden_smallest(a,b,c,d));
```

Listing 7.4: Code snippet from 'smallest'

RQ2. Can our clustering reasons (culprit constraints) help users to look for root causes of failures?

One advantage of our bucketing method compared to CSB and PFB approaches is its ability to provide a *semantic characterization* of the failures that are grouped together (based on the culprit constraint). The existing techniques are only capable of capturing syntactic information like the line number in the program or the state of the call-stack when the failure is triggered.

Table 7.4: Sample culprit constraints

Program	Culprit constraint
mkfifo	`(= (select arg0 #x00000001) #x5a)`
pr	`(= (select stdin #x00000009) #x09)`

Table 7.4 shows a few examples of the culprit constraints that our technique used to cluster failing tests for **mkfifo** and **pr**. In **mkfifo**, the culprit constraint can be interpreted as: *the second character in the first argument is the character 'Z'*. This is, in fact, the correct characterization of this bug in **mkfifo** as the tests in this cluster fail for the "`-Z`" option. In case of **pr**, the culprit constraint indicates that: *the tenth character of the standard input is a horizontal tab (TAB)*. The root cause of this failure is due to incorrect handling of the backspace and horizontal tab characters.

RQ3. What is the time overhead introduced by our bucketing technique over vanilla symbolic execution?

As shown in Table 7.3, in most of the subject programs the time overhead is negligible (from 0% to 5%), except in some programs where the overhead is dominated by the constraint solving time.

7.5.2 User Study

A user study was carried out with 18 students enrolled in a Software Security course (CS4239) in the National University of Singapore (NUS) to receive feedback on the usability and effectiveness of our bucketing method. Among the students, there were 14 senior undergraduate and 4 masters students. Before attending the course, they had no experience on applying bucketing techniques.The students were required to run the three bucketing techniques (our method and two others based on

Table 7.5: Responses from the user study.

Techniques	Level of Difficulty (Q1)				Usefulness (Q2)		
	Easy	Moderate	Difficult	Very difficult	Not useful	Useful	Very useful
Point of failure (PFB)	8	8	2	0	0	7	11
Stack hash (CSB)	3	13	2	0	3	8	7
Symbolic analysis (SAB)	1	9	7	1	2	4	12

call-stack and point of failure information) to cluster the found failing tests, and (primarily) answer the following questions:

Q1. Rate the level of difficulty in using the three techniques for bucketing failing tests.

Q2. To what extent do the bucketing techniques support debugging of program error?

Q3. Are the numbers of clusters generated by the bucketing techniques manageable?

The users' responses for Q1 & Q2 are summarized in Table 7.5; for example, the first cell of Table 7.5 shows that 8 of the 18 respondents found the PFB technique "Easy" for bucketing. In response to Q3, 14 out of the 18 respondents voted that the number of clusters generated by our technique is manageable.

In terms of usefulness as a debugging aid, our technique is ranked "Very Useful" by 12 of the 18 respondents. It gains a high rating for its usefulness as it provides a *semantic characterization* for each bucket (in terms of the culprit constraint), that can help users locate the root cause of failure. At the same time, we found that the main reason that they found our technique harder to use was that this characterization was shown in the form of logical formula in the SMT-LIB format—a format to which the students did not have enough exposure. We list some of the encouraging feedback we got:

- "I believe it is the most powerful of the three techniques, letting me

understand which assert are causing the crash or how it is formed."

- "It is very fine grain and will allow us to check the path condition to see variables that causes the error."

7.6 Chapter Summary

In this chapter, we have presented our symbolic analysis based bucketing method. We leverage the symbolic execution tree built by a symbolic execution engine to cluster failing tests found by symbolic path exploration. Our approach can also be implemented on symbolic execution engines like S2E [35] for clustering tests for stripped program binaries (when source code is not available). Unlike many other prior techniques, our technique should be able to handle changing of addresses when Address Space Layout Randomization (ASLR) is enabled as symbolic expressions are unlikely to be sensitive to address changes.

Chapter 8

Conclusion

8.1 Thesis summary

Fuzz testing techniques have become prominent for security vulnerability detection. For instance, SAGE white-box fuzzer [51] was used in testing of Windows 7 prior to its release, and AFL grey-box fuzzer [4], as shown in its homepage [1], has been used to discover more than 300 vulnerabilities in 148 large programs and libraries such as OpenSSL, PHP and Mozilla Firefox browser. However, given an *inadequate test suite* they are not skillful at directing the exploration to reach *given target locations* and expose program bugs in *large program binaries* that take *highly-structured* file inputs. In this thesis, we propose algorithms to circumvent the limitations. To this end, we design algorithms to enhance directed search in black-box, grey-box and white-box fuzzing techniques. Our algorithms take into account the *inadequacy* of given test suite, the *complex structures* of program inputs (e.g., the presence of optional data chunks, integrity checks like checksum), the *incompleteness* of program structure (e.g., control flow graph) lifted from binaries, and also the *complexity* of the program under test (e.g.,

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

multi-module design). Moreover, being aware of the overwhelming number of failing tests could be generated during fuzzing process, we also develop a fine-grained bucketing technique to effectively manage and group the tests to ease the debugging phase.

In particular, we have made the following contributions in this thesis.

- **Directed search algorithm in white-box fuzzing**. Given a (potentially) crashing location, our algorithm, which is composed by several heuristics, systematically directs the search towards the location and reasons about the crash condition to generate crash-triggering input(s). The algorithm works with real-world multi-module (stripped) binary programs like Adobe Reader and Windows Media Player.

- **Combination of model-based black-box and directed white-box fuzzing**. Such novel combination allows to exploit the best of both worlds – model-based black-box is good at generating whole chunk(s) of data while white-box fuzzing is skillful at reasoning about values of data fields – to handle missing data chunk problem in the presence of inadequate test suite.

- **Directed coverage-based grey-box fuzzing**. The integration of Simulated Annealing – a Markov Chain Monte Carlo approach – into coverage-based grey-box fuzzing (CGF) allows CGF to direct the exploration towards a given set of target locations without any expensive program analysis at run-time. Required analysis is done at compile time. To the best of our knowledge, we develop the first multiple-target search-based software testing technique where the single objective is to generate an input that exercises as many of the given targets as possible.

- **Fuzzing framework and evaluation**. The evaluations on two applications of directed fuzzing – crash reproduction and patch testing for vulnerability detection – show the effectiveness and efficiency of our techniques. Hercules, MoBWF and AFLGo successfully reproduce crashes in large real-world (binary) programs (e.g., Adobe Reader, Windows Media Player, OpenSSL, Binutils etc) taking highly-structured file formats (e.g., PDF, PNG, WAV etc). Notably, AFLGo can expose the well-known HeartBleed vulnerability in OpenSSL library almost four (4) times faster than the state-of-the-art AFL fuzzer. AFLGo has discovered 14 zero-day vulnerabilities in Binutils' utilities; five (5) CVEs have been assigned to the most critical vulnerabilities.

- **Fine-grained failing tests bucketing technique**. We leverage symbolic analysis and symbolic execution tree to identify semantic "reasons" behind failures and group failing tests into "meaningful" clusters. The semantic reason makes our approach more fine-grained compared to off-the-shelf point-of-failure and call-stack-based approaches.

8.2 Future work

Our novel combination of model-based black-box fuzzing and directed white-box fuzzing has shown its effectiveness in handling highly-structured inputs. However, input model – the key input of the technique has been manually written. One possible opportunity is construct the input model automatically from a set of benign inputs. There is a rich set of relevant research on automatic grammar inference; however the research has focused on context-free-grammar which cannot

produce structured inputs having complex relationships between data chunks and data fields (e.g., checksums, size-of, count-of, length-of etc).

Another possible avenue of future work is to bring the directed fuzzing techniques to other application domains apart from file processing applications. For instance, web applications require higher level of interaction between application and users as well as application and external systems. Another domain includes programs running inside OS kernel (e.g., device drivers, file systems) which have complex dependencies on the huge kernel code base.

Bibliography

[1] Coreutil benchmarks. http://www.gnu.org/software/coreutils/coreutils.html.

[2] Exploit-db benchmarks. https://www.exploit-db.com/.

[3] Specification of the DEFLATE Compression Algorithm. https://tools.ietf.org/html/rfc1951. Accessed: 2016-02-13.

[4] Tool: American Fuzzy Lop Fuzzer. http://lcamtuf.coredump.cx/afl/. Accessed: 2016-01-23.

[5] Tool: American Fuzzy Lop Fuzzer. https://github.com/mboehme/aflfast. Accessed: 2016-01-23.

[6] Tool: Codesonar static analysis. https://www.grammatech.com/products/codesonar. Accessed: 2016-11-20.

[7] Tool: IDA multi-processor disassembler and debugger. https://www.hex-rays.com/products/ida/. Accessed: 2016-04-04.

[8] Tool: LLVM LibFuzzer. http://llvm.org/docs/LibFuzzer.html. Accessed: 2016-01-23.

[9] Tool: Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/. Accessed: 2016-01-23.

[10] Tool: Peach Fuzzer Platform (Input Model). http://community.peachfuzzer.com/v3/DataModeling.html. Accessed: 2016-01-23.

[11] Tool: SPIKE Fuzzer Platform. http://www.immunitysec.com. Accessed: 2016-01-23.

[12] Website: Reference.com. https://www.reference.com/technology/. Accessed: 2016-11-20.

[13] Website: wired.org. https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/. Accessed: 2016-11-20.

[14] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012.

[15] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[16] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proc. 36th International Conference on Software Engineering*, June 2014.

[17] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 12–22. ACM, 2011.

[18] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, October 2007.

[19] Nikolaj Bjorner and Anh-Dung Phan. vz - maximal satisfaction with z3. In Temur Kutsia and Andrei Voronkov, editors, *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, volume 30 of *EPiC Series in Computing*, pages 1–9, 2014.

[20] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The essence of computation. chapter Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software, pages 85–108. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[21] Hanno Böck. Wie man heartbleed hätte finden können. *Golem.de*, April 2015. http://www.golem.de/news/fuzzing-wie-man-heartbleedhaette-finden-koennen-1504-113345.html (DE); https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html (EN).

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, 2016.

[23] D. Brumley, Hao Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 311–325, July 2007.

[24] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification*, July 2011.

[25] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[26] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

[29] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[30] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[31] Cristian Cadar and Hristina Palikareva. Shadow symbolic execution for better testing of evolving software. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 432–435, New York, NY, USA, 2014. ACM.

[32] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 164–181. Springer Berlin Heidelberg, 2013.

[33] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of*

the 2012 IEEE Symposium on Security and Privacy, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[34] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[35] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278. ACM, 2011.

[36] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium*, Aug 2011.

[37] Chia Yuan Cho, V. D'Silva, and D. Song. Blitz: Compositional bounded model checking for real-world programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 136–146, Nov 2013.

[38] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[39] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1084–1093, Piscataway, NJ, USA, 2012. IEEE Press.

[40] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[41] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[42] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, 2011.

[43] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, February 2013.

[44] David Freedman. *Statistical models : theory and practice*. Cambridge University Press, Cambridge New York, 2009.

[45] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, 2009.

[46] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484. IEEE Computer Society, 2009.

[47] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[48] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54. ACM, 2007.

[49] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215. ACM, 2008.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[51] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*, 2008.

[52] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, Dec 2015.

[53] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer

boundary violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, 2013.

[54] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, March 2010.

[55] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[56] Yue Jia. Hyperheuristic search for sbst. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, SBST'15, pages 15–16, 2015.

[57] Yue Jia, M.B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1 of *ICSE'15*, pages 540–550, 2015.

[58] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484. IEEE Press, 2012.

[59] Wei Jin and Alessandro Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.

[60] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering*, pages 1–34, 2016.

[61] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. Automatic and lightweight grammar generation for fuzz testing. *Comput. Secur.*, 36:1–11, July 2013.

[62] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 486–493, Washington, DC, USA, 2011. IEEE Computer Society.

[63] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[64] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Form. Asp. Comput.*, 27(3):573–609, May 2015.

[65] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[66] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–204, 2012.

[67] Language. LLVM Compiler Infrastructure. http://llvm.org/. Accessed: 2016-02-13.

[68] Eric Larson and Todd Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, pages 9–9, 2003.

[69] Wei Le. Segmented symbolic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 212–221, Piscataway, NJ, USA, 2013. IEEE Press.

[70] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 83–93, 2008.

[71] Link. AFL - Pulling Jpegs out of Thin Air, Michael Zalewski. `https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`, 2017. Accessed: 2016-03-26.

[72] Link. AFL binary instrumentation. `https://github.com/vrtadmin/moflow/tree/master/afl-dyninst`, 2017. Accessed: 2016-03-26.

[73] Link. Jenkins - Continuous Integration Platform. `https://jenkins.io/`, 2017. Accessed: 2017-01-13.

[74] Link. Oss-fuzz - continuous fuzzing for open source software. `https://github.com/google/oss-fuzz`, 2017. Accessed: 2017-01-13.

[75] Link. Search engine for the internet of things – devices still vulnerable to Heartbleed. `https://www.shodan.io/report/89bnfUyJ`, 2017. Accessed: 2016-03-26.

[76] Link. Zzuf: multi-purpose fuzzer. `http://caca.zoy.org/wiki/zzuf`, 2017. Accessed: 2017-01-13.

[77] Chao Liu and Jiawei Han. Failure proximity: A fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 46–56, New York, NY, USA, 2006. ACM.

[78] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools.

[79] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[80] Kin-Keung Ma, Khoo Yit Phang, JeffreyS. Foster, and Michael Hicks. Directed symbolic execution. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *Lecture Notes in Computer Science*, pages 95–111. Springer Berlin Heidelberg, 2011.

[81] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245. ACM, 2013.

[82] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[83] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[84] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. Automatically identifying known software problems. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, ICDEW '07, pages 433–441, Washington, DC, USA, 2007. IEEE Computer Society.

[85] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.

[86] Brian S Pak. *Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution.* PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2012.

[87] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[88] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 891–901, 2015.

[89] Andreas Podelski, Martin Schäf, and Thomas Wies. *Classifying Bugs with Interpolants*, pages 151–168. Springer International Publishing, Cham, 2016.

[90] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013.

[91] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[92] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 186–196, New York, NY, USA, 2010. ACM.

[93] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications, 2010.

[94] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 225–236, 2009.

[95] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[96] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS '16*, pages 1–16, 2016.

[97] Tool. LibPNG Library. http://www.libpng.org/pub/png/libpng.html. Accessed: 2016-02-13.

[98] Tool. Qemu Emulator. http://wiki.qemu.org. Accessed: 2016-02-13.

[99] Tool. Video Lan Client (VLC). http://www.videolan.org/index.html. Accessed: 2016-02-13.

[100] Andrs Vargha and Harold D. Delaney. A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[101] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 497–512, 2010.

[102] Xiaoyin Wang, Lingming Zhang, and Philip Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 199–210, 2015.

[103] Website. The Heartbleed Bug. http://heartbleed.com/. Accessed: 2016-02-13.

[104] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001.

[105] Xusheng Xiao, Sihan Li, Tao Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 246–256, 2013.

[106] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 61–72, New York, NY, USA, 2016. ACM.

[107] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 321–334. ACM, 2010.