Towards Resource-Aware Security Testing of Software

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Sang Kil Cha

B.S., Electrical Engineering, Korea University
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Thesis Committee:

Dr. David Brumley, Chair
Dr. Lujo Bauer
Dr. David Molnar
Dr. Vyas Sekar

Carnegie Mellon University
Pittsburgh, PA

August 10, 2015

For my daughter, Jaen.

Abstract

As software permeates every facet of life, it is imperative to assure the safety of software systems. Software vulnerabilities—exploitable software bugs—allow an attacker to destroy privacy, steal identities, and even extort money from victims. Therefore, software bugs must be discovered before an attacker can exploit them.

This dissertation presents our work on mutational fuzzing, a software testing technique for finding software bugs. Specifically, we argue that the efficiency of mutational fuzzing can drastically change depending on its parameters, and thus, automatic parameter optimization can help in improving the fuzzing efficiency. We validate this argument by designing, implementing, and evaluating several systems that employ novel techniques optimizing parameter selection for mutational fuzzing. Our specific contributions are that (1) we precisely define fuzzing and its parameter space; (2) we analytically study the effectiveness of mutational fuzzing in terms of bug finding probability; (3) we then address three strategies in optimizing mutational fuzzing over the parameter space in terms of the number of bugs found; and (4) we finally show a post-fuzzing strategy that enables prioritizing security-relevant bugs under limited resources.

Acknowledgments

I am deeply indebted to my advisor David Brumley for his support. I came to the USA as a master student without a clue. I struggled with courses due to language barrier and cultural differences. Fortunately, I met David in one of his courses. He rekindled my passion in computer hacking and security research. I would like to say that he have turned a computer hacker into a researcher.

I am grateful to Mahadev Satyanarayanan for teaching me the spirit of being a great engineer and a researcher. He has always been my role model during my graduate years, and his courses were my favorite of all time. I am thankful to Charles P. Neuman for giving me a great opportunity to think about being a good professor. I would especially like to thank Weidong Cui for being a great mentor. His relentless passion to his work always inspires me. I would also like to thank my mentors David Andersen, Lujo Bauer, David Molnar, Marcus Peinado, and Vyas Sekar for their helpful feedback, and constant support.

Heejo Lee gave me invaluable opportunities both in Korea and in Pittsburgh. I had the privilege of teaching a short course in Korea, which gave me a lot of inspiration. We also had fruitful discussion in Pittsburgh about life and research.

Thanassis Avgerinos and Alexandre Rebert tolerated me as a collaborator and as a friend. I am incredibly fortunate to have had the opportunity to work closely with them. I will never forget the team Mayhem.

John Truelove helped me get through cultural differences. He always tried to respect me even though I could not express myself, and it was one of the reasons why I could gain confidence during my master years.

I am also grateful to my awesome colleagues: Tiffany Bao, Jonathan Burket, Peter Chapman, Nicholas Christine, Anupam Datta, Samantha Gottlieb, Ivan Jager, Jiyong Jang, Limin Jia, Minsuk Kang, Gihyuk Ko, Jonghyup Lee, Soobum Lee, Yanlin Li, Brent Lim, Yue-Hsun Lin, Matthew Maurer, Iulian Moraru, Brian Pak, Adrian Perrig, Edward Schwartz, Divya Sharma, Arunesh Sinha, Michael Stroucken, Spencer Whitman, and Maverick Woo. My apologies to the other people I may have likely forgotten as an

oversight.

In addition, I would like to thank Ramakrishna Battala and Prasanna Kumar for welcoming me into Indian culture. The first year of my master cannot be explained without my awesome Indian friends.

Finally, my deep and heartfelt gratitude goes to my wife, Yeon Yim for her support, patience, constant encouragement and love. Without her, I could not have done this.

Funding Acknowledgments

This material was supported fully or in part by grants from the National Science Foundation, the Department of Defense, the Defense Advanced Research Projects Agency, Software Engineering Institute, CyLab Army Research Office, Lockheed Martin, and Northrop Grumman as part of the Cybersecurity Research Consortium. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

> If you know the enemy and know yourself you need not fear the results of a hundred battles.
>
> —Sun Tzu, The Art of War

Attackers exploit software bugs to intrude into systems, and to install malicious software (malware) such as viruses and worms. Even a single vulnerability—an exploitable bug—in a system gives an attacker ammunition for compromising the entire system. Sadly, software bugs are far from being eradicated. Launchpad—an open-source bug database—lists more than 1 million bug reports in its bug database for 10,000 open-source applications as of 2014 [99]. Furthermore, the National Vulnerabilities Database [53] publishes thousands of novel vulnerabilities per year.

In effect, discovering unknown software bugs has become a critical mission in software security for both benign and malicious parties. On the one hand, benign software developers find vulnerabilities in order to prevent security breaches in advance. Software companies, including Facebook, Google, and Microsoft, even sponsor bug bounty programs [30] to boost well-intentioned bug finding. On the other hand, cyber criminals look for new vulnerabilities in order to compromise computer systems, and even sell them in an underground market [67].

1.1 A Vision for Securing Software

My vision is to automatically discover software vulnerabilities before an attacker grasps an opportunity to exploit them. Such an automatic technique will help developers fix security critical bugs prior to releasing their software.

One may argue that devising defense mechanisms, e.g., establishing countermeasures to exploitations, is enough to assure software security. However, most defensive methods rely on assumptions that can be invalidated in practice. For example, Control-Flow Integrity (CFI) [1] theoretically guarantees that a program execution will follow only legitimate paths. In practice, however, it may fail to protect against control-flow hijack attacks due to the incompleteness of the Control-Flow Graph (CFG): there is no efficient algorithm to obtain a complete CFG from a binary executable [73].

In this dissertation, we address the *resource-aware security testing challenge* that is to find as many software bugs as possible given limited resources and to identify security-relevance of them. While traditional software testing focuses on improving the testing techniques themselves [42, 69, 129, 137], the resource-aware security testing considers: (1) how to use the existing techniques appropriately in order to maximize their bug finding effectiveness, and (2) how to pick out exploitable software bugs. At a high level, our approach consists of two major steps. First, we find as many bugs as possible given a resource constraint. Second, we analyze the bugs that we found in the first step in order to determine whether they are exploitable.

1.2 Overview: Resource-Aware Security Testing Challenge

The primary focus of this dissertation is on *fuzzing*, a software testing technique that is often used to find security bugs in practice [116, 117, 119]. Mutational fuzzing, which is a category of fuzzing (see §2.4), is of our particular interest, since it is straight-forward to formalize its algorithm. At a high level, mutational fuzzing takes in a program and a *seed*—an input string to the program—as input, and generates a test case by mutating the seed at random. The generated test case is then used to run the program to test if it can crash the program. If so, we store the test case as a crashing input. We repeat the process until a timeout is reached.

One of the major difficulties in fuzzing is that we cannot exhaustively generate all possible test cases given limited computing resources. Resources appear in many forms such as the total time budget or simply money to buy computing power. The key challenge is how we can maximize the number of bugs found in fuzzing within the same resource bound.

We approach this problem by optimizing parameters for mutational fuzzing. Mutational fuzzing

techniques typically employ a set of parameters that can be controlled by an analyst. We observed that the efficiency of fuzzing, i.e., the number of bugs found per time, could drastically change depending on the values of fuzzing parameters. Two different sets of parameter values can cause the same fuzzing technique to explore totally different executions of the Software Under Test (SUT). Therefore, we can maximize fuzzing efficiency by automatically adjusting the values of parameters.

The key intuition is that there are several common characteristics for the outcome of mutational fuzzing, and we can exploit the characteristics to devise fuzzing strategies. First, two distinct bugs of the same SUT are typically triggered with two executions exercising two different sets of program statements. Second, if two distinct crashing inputs derived from the same seed are due to the same bug, then they have common bit positions flipped from the seed. Third, a bug arrival process of mutational fuzzing for the same SUT has diminishing returns. We can use these characteristics in order to design a parameter selection strategy that gives higher priority to parameter values that maximize the fuzzing efficiency.

In this dissertation, we describe our approaches to enhancing the performance of mutational fuzzing in terms of resource utilization. We first start by formally defining the process of fuzzing along with its parameter space. Given the formal definition of fuzzing, we design and implement several strategies for optimizing the parameter choice of mutational fuzzing in order to improve its efficiency. Finally, we address the problem of checking the exploitability of bugs found. To this end, we augment typical safety properties in symbolic execution [26, 88, 94] with an exploitability property, and find an execution path that the exploitability property holds.

This dissertation contributes to the areas of software security and software engineering. Specifically, we demonstrate the impact of parameter optimization for *mutational fuzzing* in discovering memory corruption bugs. The thesis of this work is:

> *" Realizing the characteristics of the outcome of mutational fuzzing helps in designing fuzzing strategies that optimize parameter selection and prioritize bug fixing under limited resources. "*

1.3 Fuzzing for Bug Finding

As mentioned earlier, our primary focus is on fuzzing, a.k.a. fuzz testing, used by many security practitioners [61, 86, 118, 119, 138, 141] to find security bugs. The term fuzzing is a largely overloaded term, thus, we need to precisely define what fuzzing is before discussing any new techniques. The term was originally coined by Barton Miller [116] to mean a testing strategy that runs a program under test with a series of randomly generated inputs. However, fuzzing is currently used to mean various testing techniques including dynamic symbolic execution [71] and random testing [11, 40–42]. Through out this dissertation, we use the term "fuzzing" to denote software testing techniques that perform the following two steps: (1) generating a series of inputs (or test cases), and (2) observing whether the program under test manifests bugs, i.e., crashes, when it is executed with the produced inputs.

In Chapter 2, we present a formal definition of fuzzing. We then summarize the classes of fuzzing techniques, and discuss how to analyze the effectiveness of them in terms of probability, which will serve as a foundation of formal studies that we describe in the rest of the dissertation. Techniques presented in this dissertation mainly focus on *mutational fuzzing* (§2.4).

1.4 Parameter Space Reduction for Resource-Aware Fuzzing

Fuzzing typically runs with a series of different parameters. Unfortunately, however, the choice of parameters is largely manual and arbitrary in practice because a parameter can potentially have an infinite number of values. For example, a seed—typically a well-structured input—is used as a parameter for mutational fuzzing algorithms [86, 98], and one can choose any arbitrary size of input of any value to be a seed. Even though we fixed the size of the input, simple enumeration requires 2^N number of inputs, where N is the number of input bits.

In Chapter 3, we investigate several seed reduction techniques for mutational fuzzing. The question is, given a thousand of seed inputs, how we can efficiently select a subset of them to maximize the number of bugs found? Previous research indicates that increased code coverage [60] tends to find more bugs. Several papers [2, 8], presentations from well-respected computer security professionals [117, 118], as well as fuzzing tools such as Peach [61], indeed suggest using

executable code coverage as a seed selection strategy. We focus on how to mathematically formulate and reason about seed selection algorithms. Specifically, we formalize the problem as a set cover problem and check whether this formulation indeed helps in maximizing the total number of bugs found during a campaign of mutational fuzzing.

1.5 Parameter Inference for Resource-Aware Fuzzing

Our parameter reduction technique examines every candidate value in order to decide a subset of parameter values to use. However, trying out every potential parameter value can be inefficient in practice. A natural question arises: can we reduce the parameter space without investigating all potential parameter values?

In Chapter 4, we tackle this problem with respect to a fuzzing parameter called the mutation ratio—the rate between the number of bit positions to flip and the number of total bits in a seed. The key challenge in reducing the number of parameter values for the mutation ratio is that there is no single representative program execution for a given mutation ratio. Suppose we are given M different N-bit seeds. In case of seed reduction, we can simply run M different seeds to compare the quality of them. However, in case of mutation ratio reduction, we need to consider at least $M \times \binom{2^N}{\lfloor N \cdot r \rfloor}$ executions for each different mutation ratio r.

Existing mutational fuzzers employ several techniques to reduce the number of mutation ratios to consider. First, `zzuf` [98] lets an analyst choose the mutation ratio based on their expertise. It runs with either a single or a range of mutation ratios, but the analyst must specify them. Second, if not manual, the mutation ratios are derived non-adaptively, regardless of the program under test. BFF [86], for instance, splits a set of all possible nonzero mutation ratios into a predefined set of intervals, and performs scheduling over the intervals. FuzzSim [150] and `zzuf` use a predefined mutation ratio if a user does not specify a value. AFL (American Fuzzy Lop) [156] also employs several bit-flipping mutation strategies that only mutate a fixed number of bits, e.g., it flips only a single bit at random, regardless of the program under test.

On the other hand, we tackle this challenge by directly inferring a "good" mutation ratio. We observed that crashing inputs generated from the same seed in mutational fuzzing share some common properties. If two distinct crashing inputs derived from the same seed trigger the same bug,

then they have common bit positions flipped from the seed. Furthermore, there are bit positions in the seed that must not be flipped in order to trigger the crash. We use these common properties between crashing inputs to formulate the probability of seeing crashing inputs in fuzzing, and devise a way to infer a mutation ratio that maximizes it. Our results show that inferring a mutation ratio helps in improving the efficiency of fuzzing.

1.6 Resource-Aware Fuzzing with Dynamic Parameter Scheduling

With our parameter reduction and inference techniques, we only need to consider a subset of parameter values for fuzzing. However, we still need to decide how much time to use for fuzzing with each of the parameter values because we are under a limited resource, i.e., time budget. We note that fuzzing under constrained resources is akin to gambling with players wagering their resources on different fuzzing machines. Each machine in the game is configured with a distinct set of parameter values and outputs a random reward (bug) from a distribution for a given time resource. The question then becomes how much time should we allocate for each machine to maximize the fuzzing efficiency?

We tackle this problem by mathematically modeling the process of fuzzing. Since the number of bugs that can be found for each machine is finite, we can view repeated fuzzing runs using the same machine as a bug arrival process that has diminishing returns. From this observation, we statistically estimate an upperbound probability to discover remaining bugs for every machine. We then allocate resources to each of the machines to maximize the fuzzing efficiency based on the estimated upperbound probabilities.

Chapter 5 presents several scheduling algorithms over given sets of parameter values. Unlike previous works [34, 71, 122] that have mainly focused on how to improve bug finding techniques themselves, our focus is on how to use the existing techniques effectively by choosing a "good" set of parameter values.

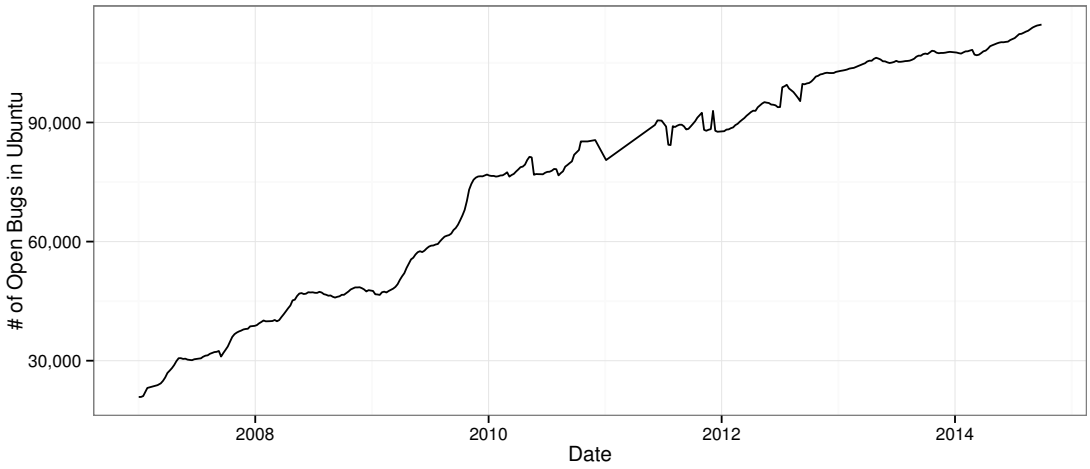


Figure 1.1: The number of open bugs in Ubuntu over time.

1.7 Resource-Aware Bug Prioritization

Fuzzing typically generates thousands of distinct crashing inputs and investigating each crashing input is primarily a manual process, which is error-prone. Unfortunately, the number of bugs to fix usually exceeds the number of developers dedicated for fixing them. For example, the number of open bugs—bugs that are reported, but not fixed—in Ubuntu keeps growing even if many developers strive to correct them. Figure 1.1 illustrates the number of open bugs in Ubuntu from January 2007 to September 2014. Of course, not all the bugs are security critical. Some of the open bugs in Ubuntu are just feature requests or aesthetic bugs that do not affect the software security.

An immediate research question that arises after a fuzzing campaign is, how can we prioritize fixing bugs that we found given limited resources? Out of thousands of bugs that are currently listed in [99], which of them should be fixed first? We note that not all bugs are equivalent. Some bugs lead to critical security breaches, but others are not. As a result, we automatically identify which bugs are subject to control-hijack attacks, thus, must be security critical.

As the first step toward the problem, we demonstrate an *automatic exploit generation* (AEG) technique that partially verifies whether a given bug is exploitable [14, 15, 38] in Chapter 6. The crux of the technique is to encode an exploitability property—the position of attack code, and the value of overwritten addresses, and so forth—and to find a program path where the exploitability property holds. AEG is sound. The exploitable test cases generated by AEG lead to a control-hijack attack. However, AEG is not complete. It does not handle all possible exploitation classes. Therefore, AEG

may falsely report a bug to be not exploitable even if it is.

1.8 Summary of Contributions

In this work, we design several techniques to tackle the *resource-aware security testing* challenge. Memory corruption bugs that lead to program crashes via segmentation fault are of particular interest because they are the root cause of most software vulnerabilities [145]. Our techniques are implemented and evaluated on black-box mutational fuzzing [113, 138]. This dissertation makes the following high-level contributions.

1. A formal definition and an analytic framework of fuzzing techniques.
2. A mathematical formulation of fuzzing parameter (seed) reduction algorithms.
3. A novel approach in inferring a fuzzing parameter (mutation ratio) from a program execution by leveraging a white-box analysis technique.
4. An investigation of various online scheduling algorithms that dynamically allocate time for given fuzz parameters in order to maximize the number of attainable bugs.
5. A novel bug prioritization strategy, called automatic exploit generation, that helps identifying security relevance (exploitability) of bugs.

Chapter 2

Theory of Fuzzing

> The original work was inspired by being logged on to a modem during a storm with lots of line noise. And the line noise was generating junk characters that seemingly were causing programs to crash. The noise suggested the term 'fuzz'.
>
> —Barton Miller

Fuzzing is a class of automated software testing [123]. The term "fuzz" was first used by Miller *et al.* [116] in the early 90s to describe a technique that feeds in random inputs to the Software Under Test (SUT) until encountering a crash. Since then, the term has been semantically overloaded to refer to various techniques that execute the SUT using a series of inputs, called test cases, in order to identify potential software bugs [71, 138]. Consequently, fuzzing nowadays refers to a variety of testing methodologies including random testing [11, 42, 129], mutational fuzzing [138, 150], dynamic symbolic execution [34, 71] and more. The details vary, but fuzzing techniques have the common theme: fuzzer executes the SUT with a series of concrete inputs.

Fuzzing is attractive because of its dynamic nature. There is no need to simulate dynamic data structures and environment as in static analyses, and the result of fuzzing is sound: when it finds a bug, then it is indeed a bug. Consequently, fuzzing has become a de facto standard in bug finding. Software companies such as Adobe [144], Google [142], and Microsoft [87] include fuzzing as a mandatory step in their development life cycle.

In this chapter, we formally define fuzzing and characterize various fuzzing techniques. We first start by defining several necessary terminologies. We then define the fuzzing algorithm, which

serves as a foundation for scientific studies that we present throughout this dissertation. Following the definition we demonstrate several analytic studies on fundamental fuzzing methodologies including random fuzzing and mutational fuzzing.

2.1 Terminology

Software bug (a.k.a. software error, fault, or flaw) [17] indicates a design or implementation mistake in a program that is introduced by one or more programmers. Sometimes a bug does not yield any noticeable effect to the program behavior, but other times a bug causes an unexpected result or behavior such as a program crash. In the worst case, a bug enables a remote attacker to hijack the control flow of the program, and lets the attacker to run arbitrary code on behalf of the victim [139]. We often call such bugs vulnerabilities.

Fuzzing involves executing a program under test with a series of concrete inputs. For each of the program executions, a fuzzer determines whether the program produced a correct result or not. We often call such an inspection process as Execution Monitoring (EM) or runtime monitoring [19]. Ideally, there should be a perfect *test oracle* [3, 18] that specifies what the output of the program should be for all possible inputs in order to find all potential bugs. In practice, however, only a specific subset of the bugs can be examined by EM.

We formulate the fuzzing process in terms of EM-enforceable safety properties as Schneider *et al.* [134]. An EM-enforceable safety property is a *safety property* [4] that bases its decision only on the past (including the current) program execution. We view fuzzing as the process of finding a set of inputs that violates an EM-enforceable safety property. Since our definition of bug finding relies on a safety property, which is enforceable by an individual execution in isolation, we may miss some bugs such as information leakages that can be discovered only with *safety policies*[1].

Let a program execution be a sequence of instructions. We represent a universe of all possible sequences as Ψ. Let Φ be the universe of all possible programs. Given a program $p \in \Phi$, Σ_p denotes a set of all possible program executions for the program p, where $\Sigma_p \subseteq \Psi$. We also let $\sigma_p(i)$ to denote an execution of p using i as an input. Since a program execution is determined by an input,

[1] Notice that this does not mean we miss all information leakages. Since we are monitoring only an individual execution at a time, we cannot determine implicit information leakage based on multiple execution paths. However, we can still identify an explicit information flow, e.g., an information leakage from a sensitive data source to an untrusted destination.

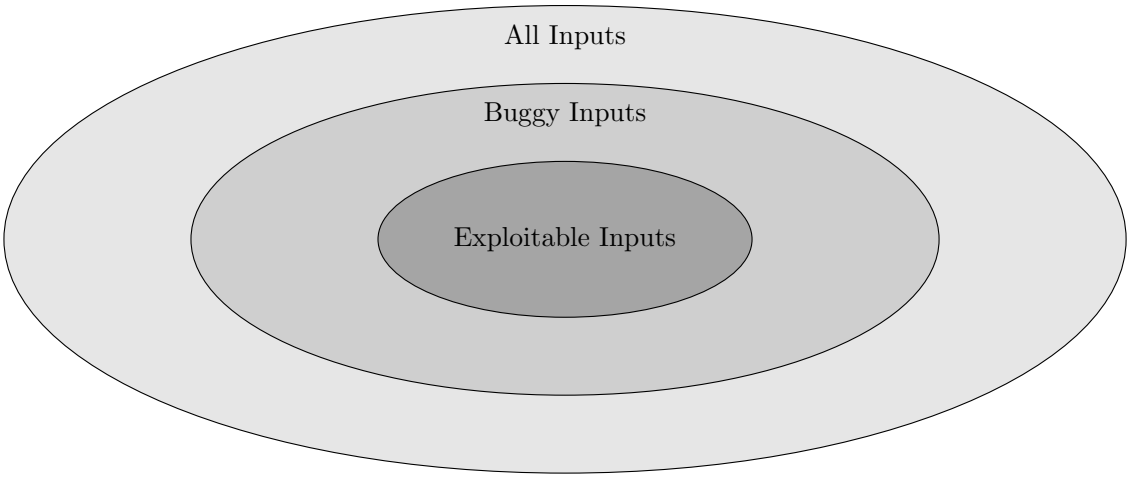


Figure 2.1: Inputs in the input space $\mathcal{I}_N$ for a program under test p. There is a subspace of $\mathcal{I}_N$ that contains only buggy inputs for the program, and there is another subspace of the subspace that contains only exploitable inputs for the program.

$\Sigma_p = \{\forall i.\sigma_p(i)\}$. An EM-enforceable safety property π is a predicate on a program execution for the program p, which returns false for a safety violation, or true otherwise. For example, $\pi(\sigma_p(i))$ is false when there is a safety violation detected by EM for the program p when it is executed with the input i. We also let Π to denote a set of all possible safety properties that are EM-enforceable. We say a program satisfies a safety property if and only if it holds true for all possible executions of the program:

$$\forall i, \sigma_p(i) \in \Sigma_p : \pi(\sigma_p(i)) = \text{true}.$$

We let an input, a.k.a. test case, be a bit string. In our model, each input has a fixed length of N bits. An input space $\mathcal{I}_N$ denotes the universe of all possible inputs of size N bits. Therefore, $|\mathcal{I}_N| = 2^N$. We call inputs that violate one or more safety properties as *buggy inputs*.

Definition 1 (*Buggy Input*)**.** An input $i \in \mathcal{I}_N$ is a buggy input for a program p, if and only if

$$\exists \pi \in \Pi : \pi(\sigma_p(i)) = \text{false}.$$

Among the buggy inputs, there can be a subset of them that leads to a control-flow hijack attack [139] while violating the safety properties. We call such inputs *exploitable inputs*, or *exploits* for short[2]. Figure 2.1 represents the subset relationship between all inputs, buggy inputs and ex-

[2] In this dissertation, we use the term "exploit" to mean a control-flow hijack exploit that allows an attacker to run any arbitrary code. Considering other classes of exploits such as information leakage is outside the scope of this dissertation.

ploitable inputs.

Many buggy inputs can correspond to a single bug (fault) in a program. Suppose a program takes in an integer input, and the program crashes whenever we supply a negative integer as an input. In this case, any negative integer is indeed a buggy input that triggers the same bug in the program. A natural question is: which bug does a buggy input correspond to? We often call the process of determining a corresponding bug from a given buggy input as *triaging*. We define a triaging function TRIAGE that takes in a program execution and a safety property, and returns either a bug identifier when the safety property violates, or $\perp$ if otherwise. For instance, if a program p violates a safety property π when we execute p with a test case i, then $\textsc{Triage}(\pi, \sigma_p(i)) = b$, where b is a number that uniquely identifies the corresponding bug.

We use a subscript to specify a bit position in an input. For example, s_1 means the first bit of the input s. We denote the Hamming distance [80]—the number of bit differences in two input strings—between input i and j in the input space by $\delta(i, j)$. Given an input i, we let a K-neighbor of i—denoted by $\mathcal{N}_K(i)$—be a set of inputs such that every input in the set has the same Hamming distance K from i.

Definition 2 (*K-Neighbor*)**.** A K-neighbor of an input i of length N is an input whose Hamming distance from i is K. We denote the set of all K-neighbors of i by $\mathcal{N}_K(i)$:

$$\mathcal{N}_K(i) = \{j \in \{0,1\}^N \mid \delta(i,j) = K\}.$$

Given the above definition, observe that two sets of K-neighbors of the same input with a different value of K are disjoint from each other:

$$\forall i, 0 \leq A, B \leq N : \mathcal{N}_A(i) \cap \mathcal{N}_B(i) = \varnothing \iff A \neq B.$$

We let μ be a function that takes as input a test case and a set of bit positions, and returns a mutated test case where every specified bit is flipped (exclusive-or-ed with 1) from the given test case. For example, $\mu(s, \{3, 4\})$ is an input where both the third and the fourth bit of s are flipped, and $\delta(\mu(s, \{3, 4\}), s) = 2$.

2.2 Our Mathematical Model

We now state and justify several assumptions of our mathematical model, all of which are satisfied by typical fuzzers in practice.

Assumption 1. *Each seed input has* finite *length.*

This assumption is always satisfied when fuzzing file inputs. In practice, some fuzzers can also perform stream fuzzing, which randomly mutates each bit in an input stream with a user-configurable probability. Notice that while the expected number of randomly-mutated bits is fixed, the actual number is *not*. We do not model stream fuzzing.

Assumption 2. *We can only observe the violation of safety property that exhibits a crash. Therefore, an execution of the program can have* exactly *one of the following two possible outcomes—it either crashes (bid) or properly terminates ($\perp$).*

In essence, this assumption means we focus exclusively on finding bugs that lead to crashes. Finding logical bugs that do not lead to crashes would typically require a correctness specification of the program under test. At present, such specifications are rare in practice and therefore this assumption does not impose a severe restriction.

Assumption 3. *The outcome of an execution of the program depends* solely *on the input generated by the fuzzer.*

This assumption ensures we are not finding bugs caused by input channels not under the fuzzer's control. Since the generated input alone determines whether the program crashes or terminates properly, all bugs found during fuzzing are deterministically reproducible. In practice, inputs that do not cause a crash in downstream analyses are discarded.

2.3 Fuzzing

A fuzzing algorithm takes in a program and a set of other parameters as input. We call a set of parameters including the program under test as *fuzz configuration*. A fuzz configuration always include a program because any algorithm needs to know which program to test. However, types

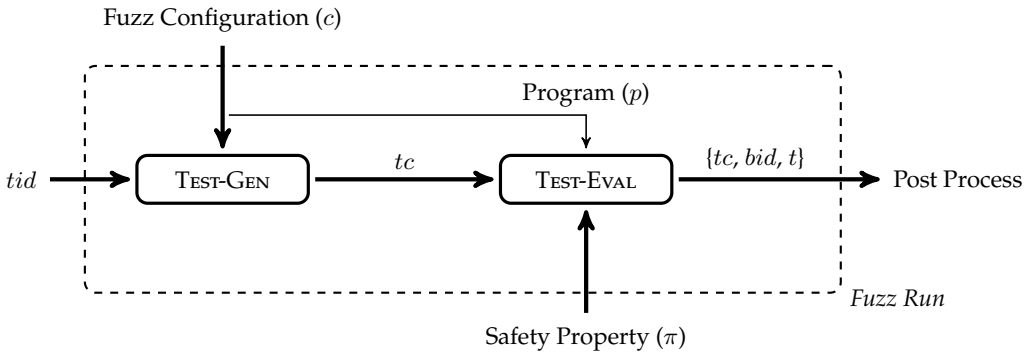


Figure 2.2: A fuzz run consists of the two consecutive algorithms (Test-Gen and Test-Eval).

of fuzzing configurations may differ for different fuzzing algorithms. For example, some fuzzing algorithms do not require any parameters other than a program, but other algorithms such as mutational fuzzing require a *seed* as a parameter (see §2.4).

We now formally define *fuzz campaign* as a single-threaded function (Fuzz) that takes in a set of fuzz configurations $\mathbb{C}$, a timeout T, and a safety property π as input. Fuzz outputs a sequence of log entries $\mathbb{L}$ for each bug found during fuzzing:

$$\textsc{Fuzz}(\mathbb{C}, T, \pi) = \mathbb{L}.$$

Each fuzzing log entry consists of a 4-tuple: a corresponding buggy input to trigger the bug (tc_i), a unique identifier of the bug (bid_i), a timestamp t_i, and a fuzz configuration that is used for finding the bug c_i. Therefore, we can represent $\mathbb{L}$ as follows:

$$\mathbb{L} = \{(tc_1, bid_1, t_1, c_1), \cdots, (tc_n, bid_n, t_n, c_n)\}.$$

In our model, Fuzz consists of a series of *fuzz runs*. A fuzz run comprises two algorithms, namely Test-Gen and Test-Eval, which behave as follows.

Test-Gen: $(tid : \mathbb{N},\ c : \{p : \Phi, \cdots\}) \rightarrow tc : \{0,1\}^N$.

Test-Gen is either a randomized or a deterministic algorithm, which takes as input a test identifier tid and a fuzz configuration c (a labeled record), and outputs a test case in the input space $\mathcal{I}_N$ where $N > 0$. Test-Gen generates an input for a program according to the fuzz configuration. The test identifier is a unique number that is given to each generated test

case: it is only meaningful in deterministic test case generation algorithms because it allows an analyst to reproduce generated test cases. The fuzz configuration c can have different types of elements depending on the underlying algorithm, e.g., mutational fuzzing has a seed as a parameter (see § 2.4).

TEST-EVAL: $(p : \Phi,\ tc : \{0,1\}^N,\ \pi : \Pi) \rightarrow (tc : \{0,1\}^N,\ bid : \{\perp, \mathbb{N}\},\ t : \mathbb{N})$.

TEST-EVAL takes as input a program p, a test case tc, and a safety property π. It executes p with tc, and returns the test case itself tc along with a unique bug identifier bid and the current timestamp t (in seconds). The bug identifier bid can either be $\perp$ if the test case does not violate the safety property π, or a unique number that identifies the corresponding bug if otherwise. We can represent bid with a triaging function TRIAGE as: $\textsc{Triage}(\pi, \sigma_p(tc))$.

Figure 2.2 illustrates the process of a fuzz run. We first generate a test case tc in TEST-GEN, and evaluate the generated test case in TEST-EVAL. After running both algorithms, we perform a post-processing step such as storing the test case to permanent storage. A fuzz campaign iterates a series of fuzz runs with given fuzz configurations. At each iteration, fuzzers execute a scheduling algorithm (FUZZ-SCHEDULE) in order to decide which fuzz configuration to use for the next fuzz run. FUZZ-SCHEDULE behaves as follows.

FUZZ-SCHEDULE: $(\mathbb{L} : \{(bid_1 : \{\perp, \mathbb{N}\},\ t_1 : \mathbb{N},\ c_1 : \mathbb{N}), \cdots\},\ \mathbb{C} : \{\{p_1 : \Phi, \cdots\}, \cdots\}) \rightarrow c : \mathbb{N}$.

FUZZ-SCHEDULE takes as input a fuzzing log $\mathbb{L}$ and a set of fuzz configurations $\mathbb{C}$, and outputs a fuzz configuration $c \in \mathbb{C}$ based on the observed fuzzing log so far. The ultimate goal of FUZZ-SCHEDULE is to allocate an optimal amount of time for each fuzz configuration in $\mathbb{C}$ in order to maximizes the fuzzing outcome (see Chapter 5).

We now formally define FUZZ with the three functions defined above. We represent the algorithm in Algorithm 2.1. FUZZ iteratively executes fuzz runs until it reaches the timeout T. Note that our definition of fuzzing differs from the notion of traditional software testing. Fuzzing is a subset of software testing that involves running the SUT with a set of generated inputs. For example, mutation testing [56, 89] is not fuzzing, because we do not allow a fuzzer to modify the SUT. Randomized differential testing [62, 111, 154]—which uses multiple compilers to compare the output of the compiled programs—are also not in the category of fuzzing, since program outputs are

Algorithm 2.1: Fuzz Campaign (Fuzz).

input : Fuzz Configurations $\mathbb{C}$, Timeout T, Safety Property π
output: Fuzzing Log $\mathbb{L}$

```
1  tid ← 0                                          // Used only for deterministic Test-Gens
2  L ← [·]                                                                  // Empty list
3  t_init ← CurrentTime() , t ← CurrentTime()        // Initialize with current timestamp
4  bid ← ⊥
5  while t − t_init < T do
6      c ← Fuzz-Schedule(L, C)
7      tc ← Test-Gen(tid, c)
8      p ← GetTargetProgram(c)          // Extract a target program from a fuzz conf.
9      tc, bid, t ← Test-Eval(p, tc, π)
10     if bid ≠ ⊥ then
11         L ← L + (tc, bid, t, c)                  // Append an entry to the fuzzing log
12     end
13     tid ← tid + 1
14 end
15 return L
```

not EM-enforceable safety properties: a program may not terminate.

2.4 Taxonomy of Fuzzing

In this section, we characterize several classes of fuzzing. Typically, fuzzing techniques split into two categories based on the ability in analyzing the internals of the program under test: either black- or white-box fuzzing.

Black-box fuzzing [20] does not see the internals of the program: it can only observe the input/output behavior of the program, and thus, it consider the program as a black-box. In some cases, such as Peach [61], it takes the input structure into account to generate more meaningful inputs. We can further split black-box fuzzing techniques into two categories based on the underlying test case generation (Test-Gen) methodologies [138]: *generation-based fuzzing* and *mutation-based fuzzing*. Generation-based fuzzing produces test cases from scratch, whereas mutation-based fuzzing generates test cases by mutating existing sample inputs, which are also called *seeds*.

On the other hand, white-box fuzzing [71] generates test cases by analyzing the program under test: it can see the internals of the program. Therefore, white-box fuzzing can utilize informed

	Generation-Based	Mutation-Based
Black-Box	**Generation-Based Black-Box** [11, 42, 48, 78, 84, 97, 132, 157]	**Mutation-Based Black-Box** [61, 86, 98, 118, 150, 156]
White-Box	**White-Box** [16, 26, 34, 44, 69–71, 88, 94, 106, 133, 137, 151]	

Table 2.1: Fuzzing taxonomy.

feedback from the program under test, and can explore the state space of the program in a systematic way. However, the overhead of generating test cases in white-box fuzzing is typically much higher than that of black-box fuzzing. For example, symbolic execution [16, 34, 71] requires dynamic instrumentation and SMT solving [55].

Previous literatures suggest a middle-ground approach, namely gray-box fuzzing [57, 141], which considers some partial knowledge about the program under test such as code coverage [120]. Strictly speaking though, gray-box approaches are also in the category of white-box fuzzing because they do access the internals of the program. Throughout the thesis, we are going to use the term white-box fuzzing to mean both gray- and white-box fuzzing.

Table 2.1 summarizes three categories of fuzzing approaches including Generation-based Black-box Fuzzing (GBF), Mutation-based Black-box Fuzzing (MBF), and White-box Fuzzing (WF).

Generation-Based Black-Box Fuzzing (GBF). GBF generates test cases from scratch. When generating test cases, GBF typically relies on structural knowledge about inputs that the program under test can take in. The structural knowledge includes grammars (grammar-based fuzzing [61, 84, 119, 132, 141, 157]) and input features (adaptive random testing [42] and partition testing [40, 41, 126]). Random testing, a.k.a. random fuzzing, [11, 59, 78] is also in this category although it does not rely on any structural knowledge. Combinatorial testing [48, 97] studies an efficient way of deriving inputs by combining a list of known inputs. Model-based testing [5, 127] uses an abstract model of the program, which is a finite state automaton, to generate test cases: since it does not directly work with the program under test, this is considered to be a black-box approach. There are also feedback-driven approaches such as [129]. Popular fuzzing tools in this category include cross_fuzz [157], jsfunfuzz [132], LangFuzz [84], and Randoop [129].

Mutation-Based Black-Box Fuzzing (MBF). Unlike generation-based fuzzing, mutation-based black-box fuzzing (mutational fuzzing for short) requires an initial seed to start generating test inputs. AFL [156], BFF [86], Radamsa [77], and `zzuf` [98] are representative fuzzers in this category. The rationale is to use a well-formed input seed to shift the fuzzer's attention to likely-to-be-valid inputs instead of spending their time on constructing a valid input from scratch. For example, a buggy input for an MP3 player is likely to be a valid MP3 file instead of being a random bit-string.

White-Box Fuzzing (WF). WF generates test cases by analyzing the program under test. For example, symbolic execution [26, 34, 70, 88, 94] is a popular way of generating test cases by systematically exploring execution paths of the program under test. Search-based testing [82, 112] typically relies on the structural information of the program such as control flow in order to direct the search. WF also uses an input seed to bootstrap its analysis. Concolic testing [16, 69, 137] is a variant of traditional symbolic execution that performs symbolic analysis along a concrete execution path. There also exists a combination of black- and white-box mutation-based fuzzing: hybrid-concolic [106] is a combination of concolic testing and mutational fuzzing. Popular white-box fuzzing tools include CUTE [137], KLEE [34], and Sage [71].

Our Scope. In this dissertation, we mainly focus on one of the fuzzing categories: MBF. In particular, we improve the efficiency of MBF using a variety of developed techniques to tackle the resource-aware security testing challenge. Although our technique relies on some techniques in WF, e.g., symbolic execution, but the primary focus is on improving the efficiency of MBF.

2.5 Fuzzing Algorithms

In this section, we investigate three fuzzing algorithms based on their test case generation methods. In particular, we define three fuzzing algorithms including random fuzzing, ball-based mutational fuzzing and surface-based mutational fuzzing. These algorithms will be used as a foundation for scientific studies of the rest of the dissertation. Our focus is on fuzzing approaches that use a randomized test case generation algorithm because it is suitable for defining the probability: deterministic algorithms require a strong assumption about the sample space in order to probabilistically analyze them. A randomized fuzzing algorithm enables us to measure the effectiveness of

fuzzing mathematically.

Random fuzzing generates random inputs to test the SUT. It falls under the category of GBF because it generates test cases from scratch. Indeed random fuzzing is the original form of fuzz testing used by Miller *et al.* [116]. Mutational fuzzers (MBF) generates test inputs by partially mutating a given *seed*[3]. They may employ a fuzzing parameter called the *mutation ratio* (r), which determines the maximum number of bit positions to flip. There can be many variants of mutational fuzzing depending on the way they mutate a seed input. In this section, we formally and succinctly define two mutational fuzzing techniques: ball-based mutational fuzzing and surface-based mutational fuzzing. Ball-based mutational fuzzing generates test cases by sampling an input from a Hamming ball centered around a given seed, whereas surface-based mutational fuzzing does the same from the surface of the Hamming ball.

2.5.1 Random Fuzzing

Random fuzzing is often called random testing [11, 59, 78] or dumb fuzzing [141]. The idea is to generate a test case uniformly at random from an input space. Due to Assumption 1, our definition of random fuzzing generates fixed-length inputs of N bits.

Definition 3 (*Random Fuzzing*)**.** Random fuzzing is a fuzz campaign that generates test cases by randomly sampling N-bit inputs from $\mathcal{I}_N$.

2.5.2 Ball-based Mutational Fuzzing

Ball-based mutational fuzzing selects test inputs from a Hamming ball centered around a given seed. More formally, ball-based mutational fuzzing generates test cases that are in a K-neighbor of an N-bit seed, where $1 \leq K \leq \lfloor N \cdot r \rfloor$.

Definition 4 (*Ball-based Mutational Fuzzing*)**.** Ball-based mutational fuzzing is a fuzz campaign that generates test cases for a given N-bit seed s by randomly sampling inputs from

$$\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_K(s).$$

[3] A seed is traditionally a well-structured input such as an MP3 file.

2.5.3 Surface-based Mutational Fuzzing

Another way to mutate a seed with respect to the mutation ratio r is to consider test cases that have the Hamming distance of exactly $\lfloor N \cdot r \rfloor$. In other words, we can consider only a single K-neighbor from a given seed instead of taking the union of all possible K-neighbors into account.

Definition 5 (*Surface-based Mutational Fuzzing*)**.** Surface-based mutational fuzzing is a fuzz campaign that generates test cases for a given N-bit seed s by randomly sampling inputs from $\mathcal{N}_{\lfloor N \cdot r \rfloor}(s)$.

In most cases, the K-neighbor of the largest K is the one that is most frequently selected from mutational fuzzing. For instance, given a 1,000-bit seed s and a mutation ratio 0.01, surface-based mutational fuzzing only uses $\mathcal{N}_{100}(s)$ to generate test cases. Notice that the cardinality of $\mathcal{N}_{100}(s)$ is $\binom{1000}{100}$, and it is the biggest K-neighbor among $0 < K \leq 100$. In this case, surface-based mutational fuzzing is equivalent to ball-based mutational fuzzing 88.9% of the time because the size of the $\mathcal{N}_{100}(s)$ covers 88.9% of the entire set: $|\mathcal{N}_{100}(s)| / \bigcup_{K=1}^{100} |\mathcal{N}_K(s)| \approx 0.889$.

2.6 Measuring the Fuzzing Efficiency

Now that we have a formal definition of fuzzing, we can model the efficiency of fuzzing in terms of probability. There are generally three ways [11] to measure the effectiveness of testing methodologies: (1) P-measure compares the probability of finding at least one buggy input using each method; (2) E-measure analyzes the expected number of buggy inputs; and (3) F-measure contrasts the expected number of test cases required to trigger at least one buggy input.

2.6.1 Random Fuzzing

Each iteration of random fuzzing is a probabilistic experiment that randomly selects a test case from an input space ($\mathcal{I}_N$). A buggy input space ($\mathcal{B}_N^p$) is a subset of the input space, which contains all buggy inputs for the program p. We call the probability of a randomly chosen input being a buggy input as a failure rate θ [40]. The failure rate of random testing θ_R for a program p is

$$\theta_R = \frac{|\mathcal{B}_N^p|}{|\mathcal{I}_N|} = \frac{|\mathcal{B}_N^p|}{2^N}.$$

Let l be the number of fuzz runs for a fuzz campaign: a fuzz campaign produces a sequence of l random test cases. Due to Assumption 2, the program under test can either crash or execute normally. We let Ω be the sample space of a fuzz run:

$$\Omega = \{\text{crash}, \text{normal}\}. \tag{2.1}$$

We assume that all test case generations in random fuzzing are independent, i.e., fuzzing always chooses a test case from the sample space *with replacement*. This means that each fuzzing iteration has exactly the same failure rate θ_R. Therefore, it is possible to have duplicated test cases during fuzzing.

P-measure. Let X_l be a random variable that represents the number of occurrences of "crash" after l number of fuzz runs. Then the probability of not finding any crash after l trials is

$$\Pr[X_l = 0] = (1 - \theta_R)^l.$$

Therefore, the P-measure—the probability of finding at least one buggy input after l fuzz runs—of random fuzzing is as follows.

$$\Pr[X_l > 0] = 1 - (1 - \theta_R)^l.$$

E-measure. From the above definition of the probability, the random variable X follows the traditional binomial distribution [64] because an outcome of an experiment is either of two values—crash and normal—which can be mapped to a success and a failure respectively. E-measure uses the expected number of buggy inputs (successes) after l fuzzing trials to examine the effectiveness as follows.

$$\mathrm{E}[X] = l \cdot \theta_R.$$

F-measure. Let Y be a random variable that represents the number of fuzz runs resulting in normal before a buggy input is found. The probability of finding a buggy input after y trials is

$$\Pr[Y = y] = (1 - \theta_R)^{y-1} \cdot \theta_R.$$

Since every event of random fuzzing is independent each other, the random variable Y follows the traditional geometric distribution [64]. Therefore, the expected number of fuzz runs until finding the first buggy input is

$$\mathrm{E}[Y] = \frac{1}{\theta_R}.$$

Example. Suppose there is a program that takes in an input file of the size 1 byte (= 8 bits), and there are only 4 buggy inputs out of 2^8 possible test cases. We also let our time budget allows us to test only 100 test cases, i.e., $l = 100$. In each fuzz run, the failure rate θ_R is simply $4/2^8$. In this case, the P-measure gives a probability

$$\Pr[X_{100} \geq 1] = 1 - (1 - \frac{4}{2^8})^{100} \approx 0.79.$$

This means, that the probability of finding at least one buggy input out of 100 fuzzing trials is approximately 0.79. The E-measure gives an expected number

$$\mathrm{E}[X_{100}] = 100 \times \frac{4}{2^8} \approx 1.56.$$

Therefore, the expected number of buggy inputs out of 100 test cases is approximately 1. Finally, the F-measure derives an expected number

$$\mathrm{E}[Y] = \frac{2^8}{4} = 64.$$

That is, the expected number of test cases required to obtain at least one buggy input is 64. Therefore, it is likely to hit a buggy input after generating 64 test cases.

2.6.2 Ball-based Mutational Fuzzing

Ball-based mutational fuzzing may miss some bugs unless the mutation ratio is 1 because it will generate test inputs from a smaller subspace of the whole input space: it can find buggy inputs *only* in a subset of $\mathcal{B}_N^p$. Let $\mathcal{B}_N^p(K, s)$ be a set of buggy inputs that are in a K-neighbor of a seed s. That is,

$$\mathcal{B}_N^p(K, s) = \{i | i \in \mathcal{N}_K(s) \wedge \pi(\sigma_p(i)) = \text{false}\}.$$

Then, a set of buggy inputs that ball-based mutational fuzzing with mutation ratio r can find from a seed s for a program p is

$$\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{B}_N^p(K, s).$$

We now define the failure rate of ball-based mutational fuzzing θ_B, which is the probability of a randomly chosen test case being a buggy input.

$$\theta_B = \frac{\left| \bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{B}_N^p(K, s) \right|}{\left| \bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_K(s) \right|} \tag{2.2}$$

We note that the only difference between random fuzzing and ball-based mutational fuzzing is that they have a different failure rate. Therefore, we can easily derive all three testing metrics from those of random fuzzing by replacing the failure rate.

P-measure.

$$\Pr[X_l > 0] = 1 - (1 - \theta_B)^l$$

E-measure.

$$\mathrm{E}[X_l] = l \cdot \theta_B.$$

F-measure.

$$\Pr[Y = y] = (1 - \theta_B)^{y-1} \cdot \theta_B.$$

2.6.3 Surface-based Mutational Fuzzing

Using the same notation as in §2.6.2, we can represent a set of buggy inputs that surface-based mutational fuzzing with mutation ratio r can find from a seed s for a program p:

$$\mathcal{B}_N^p(\lfloor N \cdot r \rfloor, s).$$

We define the failure rate of surface-based mutational fuzzing, which is denoted by θ_S with respect to a seed s and a program p as follows.

$$\theta_S = \frac{\left|\mathcal{B}_N^p(\lfloor N \cdot r \rfloor, s)\right|}{\left|\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_K(s)\right|} \tag{2.3}$$

With θ_S, we show the efficiency of surface-based mutational fuzzing using the three metrics as follows.

P-measure.

$$\Pr[X > 0] = 1 - (1 - \theta_S)^l$$

E-measure.

$$\mathrm{E}[X] = l \cdot \theta_S.$$

F-measure.

$$\Pr[Y = y] = (1 - \theta_S)^{y-1} \cdot \theta_S.$$

2.6.4 Algorithmic Implementation

At this point, it is important to discuss how to design and implement an algorithm for ball-based mutational fuzzing because it is not straightforward as it seems to be[4]. For random fuzzing, there is a trivial $O(N)$ algorithm, namely, generating a random number between 0 and $2^N - 1$ using a Pseudorandom Number Generator (PRNG) [109, 110], where each bit takes $O(1)$ time. However, designing an algorithm for mutational fuzzing is not trivial, because we want to generate only inputs in a K-neighbor of the input space for a fixed K.

Someone may argue that it is possible to use a rejection sampling to resolve the problem, but the problem is that the number of rejections can be too large to be handled when the mutation ratio gets bigger, especially when it is close to 1.0. Suppose the mutation ratio is 0.9, then it is probabilistically infeasible to get the desired number of bit flips with a rejection sampling.

[4] In fact, we are not aware of any practical fuzzers that implement either the ball-based or surface-based mutational fuzzing: they use approximated algorithms instead, which is difficult to represent with closed-form expression.

To handle this problem, we devise a two-step approach for ball-based mutational fuzzing: (1) given a set of K-neighbors, where $1 \leq K \leq \lfloor N \cdot r \rfloor$, we select a K-neighbor with a weighted probability that is proportional to the size of each neighbor; (2) we then randomly select a test case within the chosen K-neighbor.

Step 1: choose a K-neighbor. The goal of the first step is to select a K-neighbor at random, and proportionally to the size of each K-neighbor. Given an N-bit seed s and a mutation ratio r, suppose we enumerate all possible inputs from 0-neighbor to $\lfloor N \cdot r \rfloor$-neighbor in a sequential order. Then, the problem of choosing a random K-neighbor is to select a random number from 1 to $\sum_{K=0}^{\lfloor N \cdot r \rfloor} \binom{2^N}{K}$, and then check which neighbor the number resides in.

Although computing the sum of binomial coefficients is expensive, we can precompute and cache them. Let $S_0 = \sum_{K=1}^{\lfloor N \cdot r \rfloor} \binom{2^N}{K}$, $S_1 = \sum_{K=1}^{\lfloor N \cdot r \rfloor - 1} \binom{2^N}{K}, \cdots, S_{\lfloor N \cdot r \rfloor - 1} = \sum_{K=1}^{1} \binom{2^N}{K}$, and $S_{\lfloor N \cdot r \rfloor} = 1$. Then we cache the values from S_0 to $S_{\lfloor N \cdot r \rfloor}$, and check whether the random value exceeds one of the cached values. Starting from S_0, we linearly check each cached value because the likelihood of selecting each neighbor decreases.

Step 2: randomly generate an input within a K-neighbor. We note that the second step of our algorithm requires an ability to selecting test cases that have the Hamming distance K from the seed s, for any given K. This is equivalent to selecting K-bit positions from the seed and flipping them, since flipping K bits results in a test case i such that $\delta(i, s) = K$.

This is the classic random k-subset selection problem. The crux of the problem is to devise an algorithm to select K elements at random from N bit positions. A straightforward algorithm such as computing a random permutation and taking the top K elements requires $O(N)$ space and time complexity. One might consider using reservoir sampling [146] to reduce the space complexity to $O(K)$, but it still requires $O(N)$ time complexity. There are several known algorithms that have $O(K)$ space complexity while requiring only $O(K)$ time complexity in expectation [21, 125]. In this dissertation, we use Floyd-Bentley's algorithm [21, Algorithm F1] to compute the subset. This algorithm outperforms permutation-based algorithms in terms of both time and space complexity when $K < N$. Once we obtain K random bits to modify from the Floyd-Bentley's algorithm, we simply flip the selected K bits (by XORing the bits with s), and generate a new test case.

We claim that the two-step approach is theoretically the same as the ball-based mutational fuzzing, which generates a random test case from the union of K-neighbors. Both approaches have the equivalent failure rate, and thus, they have to the same result of fuzzing. Let θ'_B be the failure rate of the two-step version of ball-based mutational fuzzing. Then, the following theorem holds.

Theorem 1. $\theta_B = \theta'_B$

Proof. Let s be an N-bit seed input. In ball-based mutational fuzzing, the probability of choosing a buggy input θ_B is given by (2.2):

$$\theta_B = \frac{\left|\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{B}_N^p(K, s)\right|}{\left|\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_K(s)\right|}.$$

Since two sets of K-neighbors of the same input with a different value of K are disjoint from each other,

$$\left|\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{B}_N^p(K, s)\right| = \sum_{K=0}^{\lfloor N \cdot r \rfloor} |\mathcal{B}_N^p(K, s)|. \tag{2.4}$$

Each K-neighbor has a failure rate:

$$\frac{|\mathcal{B}_N^p(K, s)|}{|\mathcal{N}_K(s)|}.$$

Since each K-neighbor is selected proportionally to the size, the final failure rate of the modified mutational fuzzing is:

$$\begin{aligned}\theta'_B &= \sum_{K=0}^{\lfloor N \cdot r \rfloor} \left(\frac{|\mathcal{B}_N^p(K, s)|}{|\mathcal{N}_K(s)|} \cdot \frac{|\mathcal{N}_K(s)|}{\left|\bigcup_{L=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_L(s)\right|} \right) \\ &= \frac{\sum_{K=0}^{\lfloor N \cdot r \rfloor} |\mathcal{B}_N^p(K, s)|}{\left|\bigcup_{L=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_L(s)\right|}.\end{aligned}$$

Due to (2.4),

$$\theta'_B = \frac{\left|\bigcup_{K=0}^{\lfloor N \cdot r \rfloor} \mathcal{B}_N^p(K, s)\right|}{\left|\bigcup_{L=0}^{\lfloor N \cdot r \rfloor} \mathcal{N}_L(s)\right|}. \tag{2.5}$$

From (2.2) and (2.5), $\theta_B = \theta'_B$. □

Chapter 3

Parameter Reduction

> If you love life, don't waste time, for time is what life is made up of.
>
> —Bruce Lee

Fuzzers typically take in a variety of parameters—we define a set of parameters as a fuzz configuration in §2.3—that an analyst can control. Unfortunately, some parameters have too many potential values to choose. An example of such parameter is the seed. Suppose we are fuzzing an MP3 player that takes in an MP3 file as an input. There are potentially infinite MP3 files, all of which can be a valid seed parameter for fuzzing. Therefore, a natural question is which seeds should we use for fuzzing? Can we reduce the set of seed inputs to consider for fuzzing?

In this chapter, we investigate a technique for reducing the parameter space of seed for fuzzing. Specifically, we answer the following questions. First, given millions, billions, or even trillions of seed inputs, e.g., MP3 files, which should we use for fuzzing? Second, how do we measure the quality of a seed selection technique *independently* of the fuzz scheduling algorithm? For example, if we ran algorithm A on seed set S_1 and S_2, and S_1 maximized bugs, we would still be left with the possibility that with a more intelligent scheduling algorithm A' would do better with S_2 rather than S_1. Can we develop a theory to justify when one seed set is better than another with the bets possible fuzzing strategy, instead of specific examples? Finally, can we converge on a "good" seed set for fuzzing on programs for a particular file type? Specifically, if S' performs well on program P_1, how does it work on other similar applications $P_2, P_3, \cdots$?

3.1 Exploiting Characteristics of Fuzzing Outcome

We exploit the fact that two distinct bugs in a program tend to arise from two different locations of a program. Therefore, if we can find a set of seeds that maximizes the code coverage, then the likelihood of finding distinct bugs can increase. The proposed technique in this chapter is based on the following assumption.

Assumption. We assume that mutational fuzzing tends to find more bugs at the program lines that a seed can exercise. Suppose we are testing an MP3 player, and there is a bug that can only be triggered with an MP3 file compressed at a bit rate 128 Kbit/s. Then, it is more likely to find the bug with a 128 Kbit/s seed, then a 192 Kbit/s seed, although it is still possible that mutational fuzzing can correctly modify a certain input field to make the seed to be a 128 Kbit/s seed.

3.2 Seed Selection Challenge

How shall we select seed files to use for fuzzers? For concreteness, we downloaded a set of seed files S consisting of $4,912,142$ distinct files and 274 file types from Bing. The overall database of seed files is approximately 6TB. Fuzzing each program for a sufficient amount of time to be effective across all seed files is computationally expensive. Further, sets of seed files are often duplicative in the behavior elicited during fuzzing, e.g., s_1 may produce the same bugs as s_2, thus fuzzing both s_1 and s_2 is wasteful. Which subset of seed files $S' \subseteq S$ shall we use for fuzzing?

Existing research [2, 8, 58, 117] as well as tools such as Peach [61] suggest using executable code coverage as a seed selection strategy. The intuition is that many seed files likely execute the same code blocks, and such seeds are likely to produce the same bugs. For example, Miller reports a 1% increase in code coverage increases the percentage of bugs found by .92% [117]. This intuition can be formalized as an instance of the set cover problem [2, 8]. Does set cover work? Is the minimal set cover better than other set covers? Should we weight the set cover, e.g., by how long it takes to fuzz a particular seed? Previous work has shown a correlation between coverage and bugs found, but has not performed *comparative* studies among a number of approaches, nor studied how to measure optimality (§3.4).

Recall that in the *set cover problem* (SCP) [50] we are given a set X and a finite list of subsets

$\mathbb{F} = \{S_1, S_2, \ldots, S_n\}$ such that every element of X belongs to at least one subset of $\mathbb{F}$:

$$X = \bigcup_{S \in \mathbb{F}} S$$

We say that a set $\mathbb{S} \subseteq \mathbb{F}$ is a set cover of X when:

$$X = \bigcup_{S \in \mathbb{S}} S$$

The seed selection strategy is formalized as:

Step 1. The user computes the coverage for each of the n individual seed files. The output is the set of code blocks[1] executed per seed. For example, suppose a user is given $n = 6$ seeds such that each seed executes the following code blocks:

$$S_1 = \{1, 2, 3, 4, 5, 6\} \quad S_2 = \{5, 6, 8, 9\}$$
$$S_3 = \{1, 4, 7, 10\} \quad S_4 = \{2, 5, 7, 8, 11\}$$
$$S_5 = \{3, 6, 9, 12\} \quad S_6 = \{10, 11\}$$

Step 2. The user computes the cummulative coverage $X = \bigcup S_i$, e.g., $X = \{1, 2, \ldots, 12\}$ for the above.

Step 3. The user computes a set cover to output a subset $\mathbb{S}$ of seeds to use in a subsequent fuzzing campaign. For example, $\mathbb{S}_1 = \{S_1, S_4, S_3, S_5\}$ is one set cover, as is $\mathbb{S}_2 = \{S_3, S_4, S_5\}$, with $\mathbb{S}_2$ being optimal in the unweighted case.

The goal of the *minimal set cover problem* (MSCP) is to minimize the number of subsets in the set cover $\mathbb{S} \subseteq \mathbb{F}$. We call such a set $\mathbb{C}$ a *minset*. Note that a minset need not be unique, i.e., there may be many possible subsets of equal minimal cardinality. Each minset represents the fewest seed files needed to elicit the maximal set of instructions with respect to S, thus represents the maximum data seed reduction size.

In addition to coverage, we may also consider other attributes, such as speed of execution, file size, etc. A generalization of the set cover is to include a weight $w(S)$ for each $S \in \mathbb{F}$. The total cost

[1] We assume code blocks, though any granularity of unit such as instruction, function, etc. also work.

of a set cover $\mathbb{S}$ is:

$$Cost\,(\mathbb{S}) = \sum_{S \in \mathbb{S}} w(S)$$

The goal of the *weighted* minimal set cover problem (WMSCP) is to find the minimal cost cover set, i.e., $\arg\min_{\mathbb{S}} Cost\,(\mathbb{S})$.

Both the MSCP and WMSCP can be augmented to take an optional argument k (forming k-SCP and k-WSCP respectively) specifying the maximum size of the returned solution. For example, if $k = 2$ then the number of subsets is restricted to at most 2 ($|\mathbb{S}| \leq 2$), and the goal is to *maximize* the number of covered elements. Note the returned set may not be a complete set cover.

Both MSCP and WMSCP are well-known NP-hard problems. Recall that a common approach to dealing with NP-hard problems in practice is to use an approximation algorithm. An approximation algorithm is a polynomial-time algorithm for approximating an optimal solution. Such an algorithm has an *approximation ratio* $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution. The minimal set cover and weighted set cover problems both have a *greedy* polynomial-time $\ln|X| + 1$-approximation algorithm [45, 90], which is a threshold below which set cover cannot be approximated efficiently assuming NP does not have slightly superpolynomial time algorithms, i.e., the greedy algorithm is essentially the best algorithm possible in terms of the approximation ratio it guarantees [63]. Since $\ln|X|$ grows relatively slowly, we expect the greedy strategy to be relatively close to optimal.

The optimal greedy polynomial-time approximation algorithm[2] for WSCP is shown in Algorithm 3.1 as follows.

Note that the unweighted minset can be solved using the same algorithm by setting $\forall S : w(S) = 1$.

3.3 Seed Selection Algorithms

In this section we consider: the set cover algorithm from Peach [61], a minimal set cover [2], a minimal set cover weighted by execution time, a minimal set cover weighted by size, and a hotset algorithm. The first two algorithms have previously been proposed in literature; the remaining are additional design points we propose and evaluate here. We put these algorithms to the test in our

[2] Other algorithms exist to compute the weighted minset (see [50, 35-3.3]).

Algorithm 3.1: Optimal greedy polynomial-time approximation algorithm for WSCP.

```
GREEDY-WEIGHTED-SET-COVER (X, 𝔽)
1   U ← X
2   𝕊 ← ∅
3   while U ≠ ∅ do
4       S ← arg max_{S∈𝔽} |S∩U| / w(S)
5       𝕊 ← 𝕊 ∪ S
6       U ← U \ S
    end
7   return 𝕊
```

evaluation section to determine the one that yields the best results (see §3.5).

All algorithms take the same set of parameters: given $|\mathbb{F}|$ seed files, the goal is to calculate a data reduction to k files where $k \ll |\mathbb{F}|$. We assume we are given t seconds to perform the data reduction, after which the selected k files will be used in a fuzzing campaign (typically of much greater length than t). We break ties between two seed files by randomly choosing one.

- **PEACH SET.** Peach 3.1.53 [61] has a class called MinSet that calculates a cover set $\mathbb{S}$ as follows[3].

Algorithm 3.2: Minset algorithm used in Peach fuzzer 3.1.53.

```
PEACH SET (P, 𝔽)
1   𝕊 ← ∅
2   i = 1
3   for S in 𝔽 do
4       cov[i] ← MeasureCoverage( S )
5       i = i + 1
    end
6   sort( cov )                                   // sort seeds by coverage
7   for i ← 1 to |𝔽| do
8       if cov[i] \ 𝕊 ≠ ∅ then
9           𝕊 ← 𝕊 ∪ cov[i]
        end
    end
10  return 𝕊
```

[3] This is a high-level abstraction of the `Delta` and `RunCoverage` methods. We checked the current Peach implementation noticed that the sorting was removed (At Line 4 of the algorithm) in their MinSet implementation since Peach 3.1.95.

Despite having the name MinSet, Algorithm 3.2 does not calculate the minimal set cover nor a proven competitive approximation thereof.

- **RANDOM SET.** Pick k seeds at random. This approach serves as a baseline for other algorithms to beat. Since the algorithm is randomized, RANDOM SET can have high variance in terms of seed quality and performance. To measure the effectiveness of RANDOM SET, unless specified otherwise, we take the median out of a large number of runs (100 in our experiments).

- **HOT SET.** Fuzz each seed for t seconds and return the top k seeds by number of unique bugs found. The rationale behind HOT SET is similar to multi-armed bandit algorithms—a buggy program is more likely to have more bugs. In our experiments, we fuzz each seeds for 5 minutes ($t = 300$) to compute the HOT SET.

- **UNWEIGHTED MINSET.** Use an unweighted k-minset. This corresponds to standard coverage-based approaches [2, 118], and serves as a baseline for measuring their effectiveness. To compute UNWEIGHTED MINSET when k is greater than the minimum required to get full coverage, the minset is padded with files sorted based on the quality metric (coverage). We follow the same approach for TIME MINSET and SIZE MINSET.

- **TIME MINSET.** Return a k-execution time weighted minset. This algorithm corresponds to the observation in Chapter 5 that weighting by time in a multi-armed bandit fuzzing algorithm tends to perform better than the unweighted version. The intuition is that seeds that are fast to execute ultimately lead to far more fuzz runs during a campaign, and thus potentially more bugs.

- **SIZE MINSET.** Return a k-size weighted minset. Weighting by file size may change the ultimate minset, e.g., many smaller files that cover a few code blocks may be preferable to one very large file that covers many code blocks—both in terms of time to execute and bits to flip. For example, SIZE MINSET will always select a 1KB seed over a 100MB seed, all other things being equal.

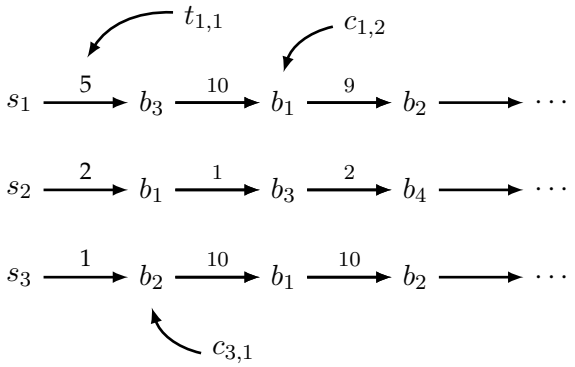


Figure 3.1: An example of output from fuzzing 3 seeds. Bugs may be found across seeds (e.g., b_1 is found by all seeds. A single seed may produce the same bug multiple times, e.g., with s_3. We also show the corresponding ILP variables t (interarrival times) and c (crash ids).

3.4 Measuring Seed Selection Quality

There are a variety of seed selection strategies, e.g., to use minset or to pick k seeds at random. How can we argue a particular seed selection strategy performs well?

One strawman answer is to run seed selection algorithm A to pick subset S_A, algorithm B to pick subset S_B. We then fuzz S_A and S_B for an equal amount of time and declare the fuzz campaign with the most bugs the winner. The fuzz campaign will incrementally fuzz each seed in each set according to its own scheduling algorithm. While such an approach may find the best seed selection for a *particular* fuzzing strategy, it provides no evidence that a particular subset is inherently better than another in the limit.

The main intuition in our approach is to measure the *optimal case* for bugs found with a particular subset of seeds. The best case provides an upper bound on any scheduling algorithm instead of on a particular scheduling algorithm. Note the lower bound on the number of bugs found for a subset is trivially zero, thus all we need is an upper bound.

To calculate the optimal case, we fuzz each seed in s_i for t seconds, recording as we fuzz the arrival rate of bugs. Given n seeds, the total amount of time fuzzing is $n * t$. For example, given 3 seeds we may have a bug b_i arrival time given by Figure 3.1.

Post-fuzzing, we then calculate the ex post facto optimal search strategy to maximize the number of bugs found. It may seem strange at first to calculate the optimal seed selection strategy after all seeds have been fuzzed at first blush. However, by doing so we can measure the quality of the seed selection strategy with respect to the optimal, thus give the desired upper bound. For exam-

ple, if the seed selection strategy picks s_1 and s_2, we can calculate the maximum number of bugs that could be found by any scheduler, and similarly for the set s_2, s_3 or any other set. Note we are calculating the upper bound to scientifically justify a particular strategy. For example, our experiments suggest to use UNWEIGHTED MINSET for seed selection. During a practical fuzzing campaign, one would not recompute the upper bound for the new dataset; instead, she would use the seed selection strategy that was shown to empirically perform best in previous tests.

Recall from §2.3, a fuzz campaign returns a sequence of 4-tuples:

$$[(tc_1, bid_1, t_1, c_1), \cdots, (tc_n, bid_n, t_n, c_n)].$$

Given that we know the ground truth, i.e., we know the return value of FUZZ when applied on every singleton in $\mathbb{F}$, we can model the computation of the optimal scheduling/seed selection across all seed files in $\mathbb{F}$. Note that the ground truth is necessary, since any optimal solution can be only computed in retrospect (if we know how each seed would perform). We measure optimality of a scheduling/seed selection by computing the maximum number of unique bugs found.

The *optimal budgeted ex post facto scheduling problem* is given the ground truth for a set of seeds and a time threshold T, automatically compute the interleaving of fuzzed seeds (time slice spent analyzing each one) to maximize the number of bugs found. The number of bugs found for a given minset gives an upper bound on the performance of the set and can be used as a quality indicator. Note that the same bug may be found by different seeds and may take different amounts of time to find. Unlike our approximated algorithm described in Chapter 5, we find an optimal schedule for a given ground truth. We observe finding an optimal scheduling algorithm is inherently an integer programming problem. We formulate finding the exact optimal seed scheduling as an Integer Linear Programming (ILP) problem [135]. While computing the optimal schedule is NP-hard, ILP formulations tend to work well in practice.

3.4.1 ILP Formulation

First, we create an indicator variable for unique bugs found during fuzzing.

$$bg_x = \begin{cases} 1 & \text{The schedule includes finding unique bug } x \\ 0 & \text{Otherwise} \end{cases}$$

The goal of the optimal schedule is to maximize the number of bugs. However, we do not see bugs, we see individual crashes arriving during fuzzing. We create an indicator variable $cr_{i,j}$ that determines whether the optimal schedule includes the j^{th} crash of seed i:

$$cr_{i,j} = \begin{cases} 1 & \text{The schedule includes crash } j \text{ for seed } i \\ 0 & \text{otherwise} \end{cases}$$

Note that multiple crashes $cr_{i,j}$ may correspond to the same bug. Crashes are triaged to unique bugs via a uniqueness function denoted by μ. In our experiments, we use stack hash [122], a non-perfect but industry standard method. Thus, if the total number of unique stack hashes is U, we say we found U unique bugs in total. The invariant is:

$$bg_x = 1 \text{ iff } \exists\, i, j : \mu(c_{i,j}) = x \tag{3.1}$$

Thus, if two crashes $cr_{i,j}$ and $cr_{i',j'}$ have the same hash, a schedule can get at most one unique bug by including either or both crashes.

Finally, we include a cost for finding each bug. We associate with each crash the incremental fuzzing cost for seed S_i to find the bug:

$$\forall i : tc_{i,j} = \begin{cases} a_{i,1} & , j = 1 \\ a_{i,j} - a_{i,j-1} & , j > 1 \end{cases}$$

where $a_{i,j}$ is the arrival time for the $cr_{i,j}$ crash, and $tc_{i,j}$ represents inter-arrival time—the time interval between the occurrences of $cr_{i,j-1}$ and $cr_{i,j}$. Figure 3.1 visually illustrates the connection between $cr_{i,j}$, bg_x and $tc_{i,j}$.

We are now ready to phrase optimal scheduling with a fixed time-budget as an ILP maximiza-

tion problem:

$$\textbf{maximize} \quad \sum_{x} bg_x$$

$$\textbf{subject to} \quad \bigvee_{i,j} . \, cr_{i,j+1} \leq cr_{i,j} \tag{3.2}$$

$$\sum_{i,j} cr_{i,j} \cdot tc_{i,j} \leq T \tag{3.3}$$

$$\bigvee_{i,j} . \, cr_{i,j} \leq bg_x \text{ where } \mu(c_{i,j}) = x \tag{3.4}$$

$$\bigvee_{x} . \, bg_x \leq \sum_{i,j} cr_{i,j} \text{ where } \mu(c_{i,j}) = x \tag{3.5}$$

Constraint (3.2) ensures that the schedule considers the order of crashes found. In particular, if the j-th crash of a seed is found, all the previous crashes must be found as well. Constraint (3.3) ensures that the time to find all the crashes does not exceed our time budget T. Constraints (3.4) and (3.5) link crashes and unique bugs. Constraints (3.4) says that if a crash is found, its corresponding bug (based on stack-hash) is found, and the next equation guarantees that if a bug is found, at least one crash triggering this bug was found.

Additionally, by imposing one extra inequality:

$$\sum_{i} cr_{i,1} \leq k \tag{3.6}$$

we can bound the number of used seeds by k (if the first crash of a seed is not found, there is no value in fuzzing the seed at all), thus getting k-bounded optimal budgeted scheduling, which gives us the number of bugs found with the optimal minset of size up to k.

3.4.2 Optimal Seed Selection for Round-Robin

The formulation for optimal budgeted scheduling gives us a best solution any scheduling algorithm could hope to achieve both in terms of seeds to select (minset) and interleaving between explored seeds (scheduling). We can also model the optimal seed selection for specific scheduling algorithms with the ILP formulation. We show below how this can be achieved for Round-Robin,

as this may be of independent interest.

Round-Robin scheduling splits the time budget between the seeds equally. Given a time threshold T and N seeds, each seed will be fuzzed for $\frac{T}{N}$ units of time. Round-Robin is a simple but effective scheduling algorithm in many adversarial scenarios as we discussed in Chapter 5. Simulating Round-Robin for a given set of seeds is straightforward, but computing the optimal subset of seeds of size k with Round-Robin cannot be solved with a polynomial algorithm. To obtain the optimal minset for Round-Robin, we add the following inequality to Inequalities 3.2-3.6:

$$\bigvee_i . \sum_j cr_{i,j} \cdot tc_{i,j} \leq \frac{T}{k} \tag{3.7}$$

The above inequality ensures that none of the seeds will be explored for more than $\frac{T}{k}$ time units, thus guaranteeing that our solution will satisfy the Round-Robin constraints. Similar extensions can be used to obtain optimal minsets for other scheduling algorithms.

3.5 Experiments

We now evaluate the overall performance of seed selection algorithms in terms of bug discovered, and answer several research questions regarding the algorithms. We start by describing our experimental setup.

Experimental Setup. All of our experiments were run on medium and small VM instance types on Amazon EC2 (the type of the instance used is mentioned in every experiment). All VMs were running the same operating system, Debian Linux 7.4. The fuzzer used throughout our experiments is the CERT Basic Fuzzing Framework (BFF) [86]. All seed files gathered for our fuzzing experiments (4,912,142 files making up more than 6TB of data) were automatically crawled from the internet using the Bing API. Specifically, file type information was extracted from the open source Gnome Desktop application launcher data files and passed to the Bing API such that files of each type could be downloaded, filtered, and stored on Amazon S3. Coverage data was gathered by instrumenting applications using the Intel PIN framework and a standard block-based coverage collection PIN tool.

3.5.1 Establishing Ground Truth

We first collect the ground truth data for fuzzing campaigns that account for every possible seed selection and scheduling. We present our methodology for selecting the target applications, files to fuzz, and parameters for computing the ground truth.

Target Applications. We selected 10 applications and 5 popular file formats: PDF, MP3, GIF, JPG and PNG for our experiments. Our program selection contains GUI and command line applications, media viewers, players, and converters. We manually mapped each program to a file format it accepts and formed 13 distinct (application, file formats) to be fuzzed—shown in Table 3.2. We selected at least two distinct command lines for each file type to test transferability (§3.5.5).

Seed Files. For each file type used by the target applications, we sampled uniformly at random 100 seed files (hence selecting $|\mathbb{F}| = 100$ for the seed file pool size) of the corresponding type from our seed file database. Note that determining ground truth for a single seed requires 12 hours, thus finding ground truth on all 4,912,142 is—for our resources—infeasible.

Fuzzing Parameters. Each of the target applications was fuzzed for 12 hours with each of the 100 randomly selected seed files of the right file type. Thus, each target application was fuzzed for 1,200 hours for a total of 650 CPU-days on an EC2 (m1.small) instance. All detected crashes were logged with timestamps and triaged based on BFF's stack hash algorithm.

Fuzzing results. BFF found 2,941 unique crashes, identified by their stack hash. BFF crashed 8 programs out of the 10 target applications. 2,702 of the unique crashes were found on one application, `mp3gain`. Manual inspection showed that the crashes were due to a single exploitable buffer overflow vulnerability that mangled the stack and confused BFF's stack-based uniqueness algorithm. When reporting our results, we therefore count the 2,702 unique crashes in `mp3gain` as one. With that adjustment, BFF found 240 bugs. Developing and experimenting with more robust, effective, and accurate triaging algorithms is an open research problem and a possible direction for future work.

Simulation. The parameters of the experiment allow us to run simulations and reason about all possible seed selections (among the 100 seeds of the application) and scheduling algorithms for a horizon of 12 hours on a single CPU. Our simulator uses our ILP formulation from §3.4.1 to compute optimal seed selections and scheduling for a given time budget. Using the ground truth, we can run simulations to evaluate the performance of hour-long fuzzing campaigns within minutes, following a replay-based fuzzing simulation strategy similar to FuzzSim.

3.5.2 Seed Selection Algorithms vs. Random Sampling

Spending resources on a seed selection algorithm is only useful if the selected seeds outperform random seed sampling (RANDOM SET). In this experiment, we compare the performance of selection algorithms as presented in §3.3 against the random sampling baseline.

All selection algorithms are deterministic, while RANDOM SET is randomized. Thus, we cannot show that RANDOM SET is always better (or worse), but we can instead compute the probability that RANDOM SET is better (or worse). To estimate the probability, we setup the following random experiment: we randomly sample a set of seeds—the size of the set is the same ($k = 10$ in our experiment for an order of magnitude reduction) as the competing reduced set—from the seed pool and measure the number of bugs found. The experiment has three possible outcomes: (1) the random set finds more bugs, (2) the random set finds fewer bugs, or (3) the random and the competitive set find the same number of bugs.

We performed 13,000 repetitions of the above experiment—1,000 for each (application, file format) tuple—and measured the frequency of each event when the optimal scheduling algorithm is employed for both. We then repeated the same experiment while using Round-Robin as the scheduling algorithm. We calculated the probability by dividing the frequency by the number of samples. Figure 3.2 summarizes the results. For instance, the left-most bar is the result for HOT SET with the optimal scheduling. You can see that HOT SET finds more bugs than a RANDOM SET of the same size with a probability of 32.76%, and it is worse with a probability of 18.57%. They find the same amount of bugs with a probability of 48.66%.

The first pattern that seems to persist through scheduling and selection algorithms (based on Figure 3.2) is that there is a substantial number of ties—RANDOM SET seems to behave as well as selection algorithms for the majority of the experiments. This is not surprising, since 3/13 (23%) of

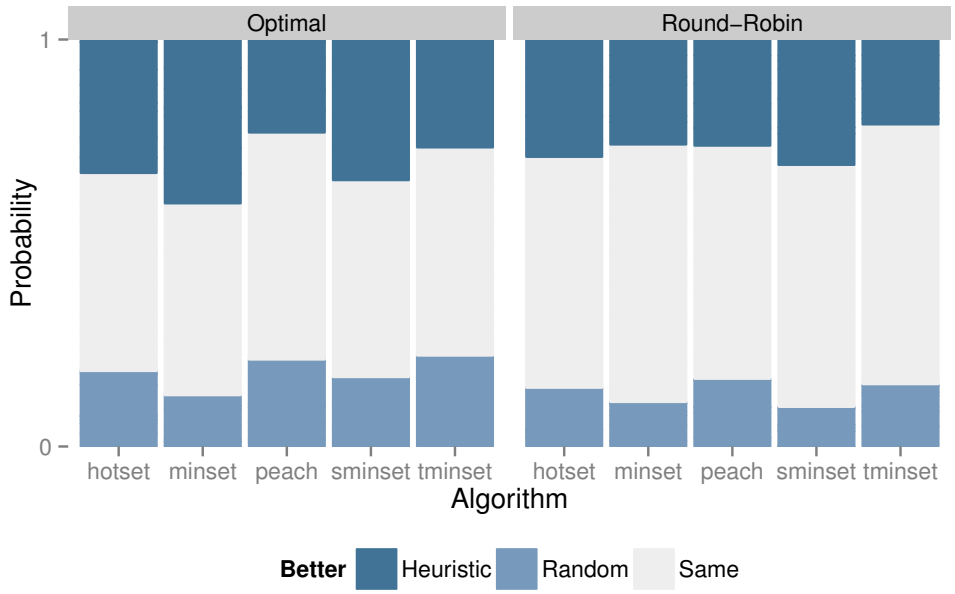


Figure 3.2: Comparing bug-finding performance of seed selection algorithms against RANDOM SET.

MinSet Algorithm	Optimal	Round-Robin
HOT SET	63.58%	67.12%
PEACH SET	50.64%	60.30%
UNWEIGHTED MINSET	75.24%	70.24%
SIZE MINSET	66.33%	75.78%
TIME MINSET	52.60%	57.62%

Table 3.1: Conditional probability of an algorithm outperforming RANDOM SET with k=10, given that they do not have the same performance (P_{win}).

our (application, file format) combinations—(mplayer, MP3), (eog, JPG), (jpegtran, JPG)—do not crash at all. With no crash to find, any algorithm will be as good as random. Thus, to compare an algorithm to RANDOM SET we focus on the cases where the two algorithms differ, i.e., we compute the conditional probability of winning when the two algorithms are not finding the same number of bugs.

We use P_{win} to denote the conditional probability of an algorithm outperforming RANDOM SET, given that they do not have the same performance. For example, for SIZE MINSET, P_{win} is defined as: P[SIZE MINSET $>$ RANDOM SET | SIZE MINSET $\neq$ RANDOM SET]. Table 3.1 shows the values of P_{win} for all algorithms for sets of size $k = 10$. We see that UNWEIGHTED MINSET and SIZE MINSET are the algorithms that more consistently outperform RANDOM SET with a P_{win} ranging from 66.33%

				Random Set		Hot Set		Unweighted Minset		Time Minset		Size Minset		Peach Set	
Files	**Programs**	**Crashes**	**Bugs**	#S	#B	#S	#B	#S	#B	#S	#B	#S	#B	#S	#B
PDF	xpdf	706	57	10	7	10	9	32	19	32	16	40	19	54	31
	mupdf	6,570	88	10	13	10	14	40	29	43	29	49	31	59	31
	pdf2svg	5,720	81	10	14	10	27	36	48	39	43	45	47	53	49
MP3	ffmpeg	1	1	10	0	10	1	11	0	11	0	22	0	19	0
	mplayer	0	0	10	0	10	0	10	0	12	0	14	0	23	0
	mp3gain	434,400	2,702	10	92	10	9	9	150	8	74	10	74	14	175
GIF	eog	9	1	10	0	10	1	29	0	27	0	43	1	44	1
	convert	72	2	10	1	10	1	13	1	14	0	24	2	22	1
	gif2png	162,302	6	10	4	10	4	16	5	17	5	29	5	33	4
JPG	eog	0	0	10	0	10	0	31	0	31	0	47	0	53	0
	jpegtran	0	0	10	0	10	0	10	0	12	0	21	0	23	0
PNG	eog	123	2	10	1	10	1	30	2	30	2	45	2	49	2
	convert	2	1	10	0	10	0	11	1	12	1	17	1	16	1
Total		609,905	2,941		132		67	278	255	288	170	406	182	462	295

Table 3.2: Programs fuzzed to evaluate seed selection strategies and obtain ground truth. The columns include the number of seed files (#S) obtained with each algorithm, and the number of bugs found (#B) with the optimal scheduling strategy.

to 75.78%. Hot Set immediately follows in the 63-67% range, and Time Minset, Peach Set have the worst performance. Note that Peach Set has a P_{win} of 50.64% in the optimal schedule effectively meaning that it performs very close to a random sample on our dataset.

Conclusion: seed selection algorithms help. With the exception of the Peach Set and Time Minset algorithms which perform very close to Random Set, our data shows that heuristics employed by seed selection algorithms perform better than fully random sampling. However, the bug difference is not sufficient to show that *any* of the selection algorithms is *strictly* better with statistical significance. Fuzzing for longer and/or obtaining the ground truth for a larger seed pool are possible future directions for showing that seed selection algorithms are *strictly* better than choosing at random.

3.5.3 Comparison

Table 3.2 shows the full breakdown of the reduced sets computed by each algorithm with the optimal scheduling algorithm. Columns 1 and 2 show the file type and program we are analyzing, while columns 3 and 4 show the total number of crashes and unique bugs (identified by stack hash) found during the ground truth experiment. The next six columns show two main statistics (in sub-

columns) for each of the seed selection algorithms: 1) the size of the set k (#S), and 2) the number of bugs (#B) identified with optimal scheduling. All set cover algorithms (Peach Set, Unweighted Minset, Time Minset, Size Minset) were allowed to compute a full-cover, i.e., select as many files as required to cover all blocks. The other two algorithms (Random Set and Hot Set) were restricted to sets of size $k = 10$.

Bug Distribution and Exploitability. The fuzzing campaign found bugs in 10/13 configurations of $\langle$program, file type$\rangle$, as shown in table 3.2. In 9/10 configurations we found less than 100 bugs, with one exception: `mp3gain`. We investigated the outlier further, and discovered that our fuzzing campaign identified an exploitable stack overflow vulnerability—the mangled stack trace can create duplicates in the stack hash algorithm. We verified the bug is exploitable and notified the developers, who promptly fixed the issue.

Reduced Set Size. Table 3.2 reflects the ability of the set cover algorithms to reduce the original dataset of 100 files. As expected, Unweighted Minset is the best in terms of reduction ability, with 278 files for obtaining full cover. Time Minset requires slightly more files (288). Size Minset and Peach Set require almost twice as many files to obtain full cover (406 and 462 respectively).

Bug Finding. The Peach Set algorithm finds the highest number of bugs (295), followed by Unweighted Minset (255), Size Minset (182) and Time Minset (170). Hot Set and Random Set find substantially fewer bugs when restricted to subsets of size up to 10. We emphasize again that bug counts are measured under optimal scheduling and thus size of the reduced set is analogous to the performance of the selection algorithm (the highest number of bugs will be found when all seeds are selected). Thus, to compare sets of seeds in terms of bug-finding ability we need a head to head comparison where sets have the same size k.

Figure 3.3 shows all selection algorithms and how they perform in terms of average number of bugs found as a function of the parameter k—the size of the seed file set. The "$\times$" symbols represent the size after which each algorithm achieves a full cover (after that point extra files are added sorted by the metric of the selection algorithm, e,g,, by coverage in Unweighted Minset). As witnessed in the comparison against Random Set, Unweighted Minset consistently performs better than other seed selection algorithms. Time Minset and Peach Set also eventually converge

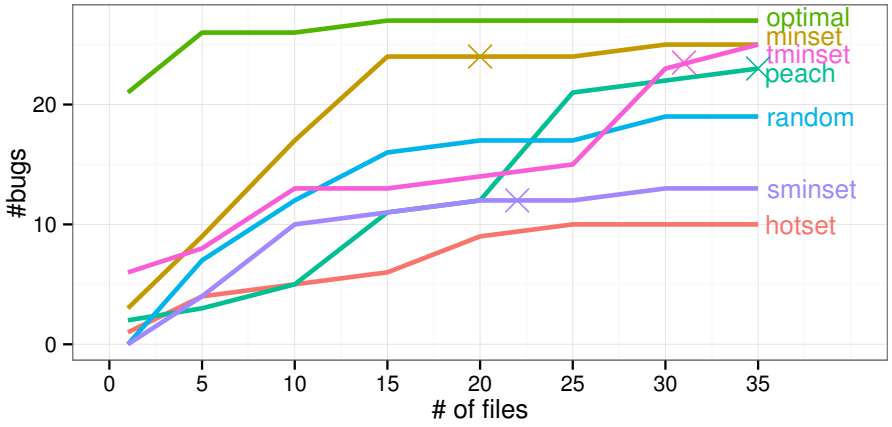


Figure 3.3: Number of bugs found by different seed selection algorithms with optimal scheduling.

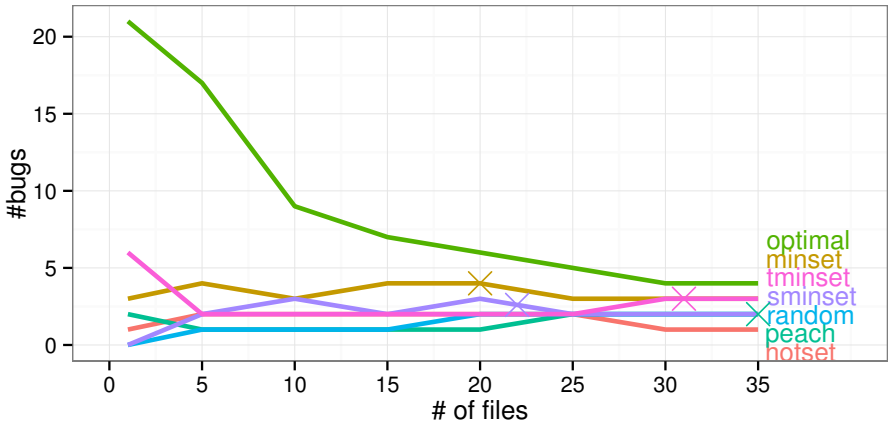


Figure 3.4: Number of bugs found by different seed selection algorithms with Round-Robin.

to the performance of Unweighted Minset under optimal scheduling, closely followed by random. Hot Set performs the worst, showing that spending time exploring *all* seeds can be wasteful. We also note, that after obtaining full cover (at 20 seed files), Unweighted Minset's performance does not improve at the same rate—showing that adding new files that do not increase code coverage is not beneficial (even with optimal scheduling).

We performed an additional simulation, where all reduced sets were run with Round-Robin as the scheduling algorithm. Figure 3.4 shows the performance of each algorithm as a function of the parameter k. Again, we notice that that Unweighted Minset is outperforming the other algorithms. More interestingly, we also note that Unweighted Minset's performance actually *drops* after obtaining full cover. This shows that minimizing the number of seeds is important; adding more seeds in Round-Robin seems to hurt performance for all algorithms.

Conclusion: Unweighted Minset performed best. Unweighted Minset outperformed all other algorithms in our experiments, both for optimal and Round-Robin scheduling. This experiment confirms conventional wisdom that suggests collecting seeds with good coverage for successful fuzzing. More importantly, it also shows that computing a minimal cover with an approximation with a proven competitiveness ratio (Unweighted Minset) is better than using an algorithm with no guaranteed competitive ratio (Peach Set).

3.5.4 Seed Reduction Usefulness

We now test the premise of using a reduced data set. Will a reduced set of seeds find more bugs than the full set? We simulated a fuzzing campaign with the full set, and with different reduced sets. We compare the number of bugs found by each technique.

Using the optimal scheduling, the full set will *always* find more, or the same amount of bugs, than any subsets of seeds. Indeed, the potential schedules of the full set is a superset of the potential schedules of any reduced set. Therefore, the full set will always better than or equal to any reduced set. To be more realistic, we use a Round-Robin schedule to test the premise.

The "$\times$" symbols on Figure 3.4 shows the unpadded size of the different selection algorithms. For those sizes, Unweighted Minset found 4 bugs on average, and the other minset algorithms found between 2.5 and 3 bugs. Fuzzing with the full set uncovered only 1 unique bug on average.

File	Application	FULL SET	UNWEIGHTED MINSET (k=10)
PDF	xpdf	53%	70%
	mupdf	83%	90%
	pdf2svg	71%	80%
MP3	ffmpeg	1%	0%
	mplayer	0%	0%
	mp3gain	95%	100%
GIF	eog	8%	0%
	convert	12%	10%
	gif2png	97%	100%
JPG	eog	0%	0%
	jpegtran	0%	0%
PNG	eog	22%	30%
	convert	2%	10%

Table 3.3: Probability that a seed will produce a bug in 12 hours of fuzzing.

We also measure the quality of seeds by looking at the average seed quality contained in that set. Our hypothesis is that a reduced set increases the average seed quality compared to the full set. To measure quality, we computed the probability of a seed producing a bug after fuzzing it for 12 hours, when the seed is picked from the full set or the UNWEIGHTED MINSET. Table 3.3 lists the results of this experiment. The UNWEIGHTED MINSET had a higher seed quality than the full set in 7 cases, while the opposite was true in 3 cases. They were tied on the 3 remaining cases.

Conclusion: Fuzzing with a reduced sets is more efficient in practice. The UNWEIGHTED MINSET outperformed the full set in our two experiments. Our data demonstrates that using seed selection techniques is beneficial to fuzzing campaigns.

3.5.5 Seed Transferability

We showed that seed selection algorithms improve fuzzing in terms of bug-finding performance. However, performing the data reduction may be computationally expensive; for instance, all set cover algorithms require collecting coverage information for all the seeds. Is it more profitable to invest time computing the minset to fuzz an efficient reduced set, or to simply fuzz the full set of seeds for the full time budget? In other words, is the seed selection worth the effort to be performed online?

We answer that question by presenting parts of our dataset. For example, our JPG bucket contains 530,727 distinct files crawled from the web. Our PIN tool requires 55 seconds (on average based on the 10 applications listed in Table 3.2) to compute code coverage for a single seed. Collecting coverage statistics for all our JPG files would take 368 CPU-days. For fuzzing campaigns shorter than a year, there would not be enough time to compute code coverage, let alone finding more bugs than the full set.

The result above indicates that, while seed selection techniques help improve the performance of fuzzing, their benefits may not outweigh the costs. It is *impractical* to spend a CPU year of computation to perform a separate seed selection for every new application that needs fuzzing.

However, recomputing the reduced set for *every application* may not be necessary if we can reuse a reduced set for every *file type*. Our intuition is a reduced set that is of high-quality for application A should also be high-quality for application B—assuming they accept the same file type. Thus, precomputing reduced sets for popular file types *once*, would allow us to instantly select a high-quality set of seed files to start fuzzing. To test transferability of reduced sets, we measure seed quality by computing code coverage achieved by a minset across programs.

Do Reduced Sets Transfer Coverage? Using the seed files from our ground truth experiment (§3.5.1) we measured the cumulative code coverage achieved in each configuration (program and file format) with reduced Unweighted Minsets computed on all other configurations (for a total of $13 \times 13 \times 100$ coverage measurements). All measurements were performed on a c1.medium instance on amazon.

Figure 3.5 is a heat map summarizing our results. The configurations on the bottom (x-axis) represent all computed Unweighted Minsets, while the configurations on the left (y-axis) represent the configurations tested. Darker colors indicate that the selected Unweighted Minset obtains higher coverage. For example, if we select the `pdf.mupdf` minset from the x-axis, we can see how it performs on all the other configurations on the y-axis. For instance, we notice that `pdf.mupdf` minset performs noticeably better on 5 configurations: `pdf.mupdf` (expected since this is the configuration on which we computed the minset), `pdf.xpdf` and `pdf.pdf2svg` (expected since these applications also accept `pdfs`), and interestingly `png.convert` and `gif.convert`. Initially we were surprised that a PDF minset would perform so well on `convert`; it turns out that this result is not

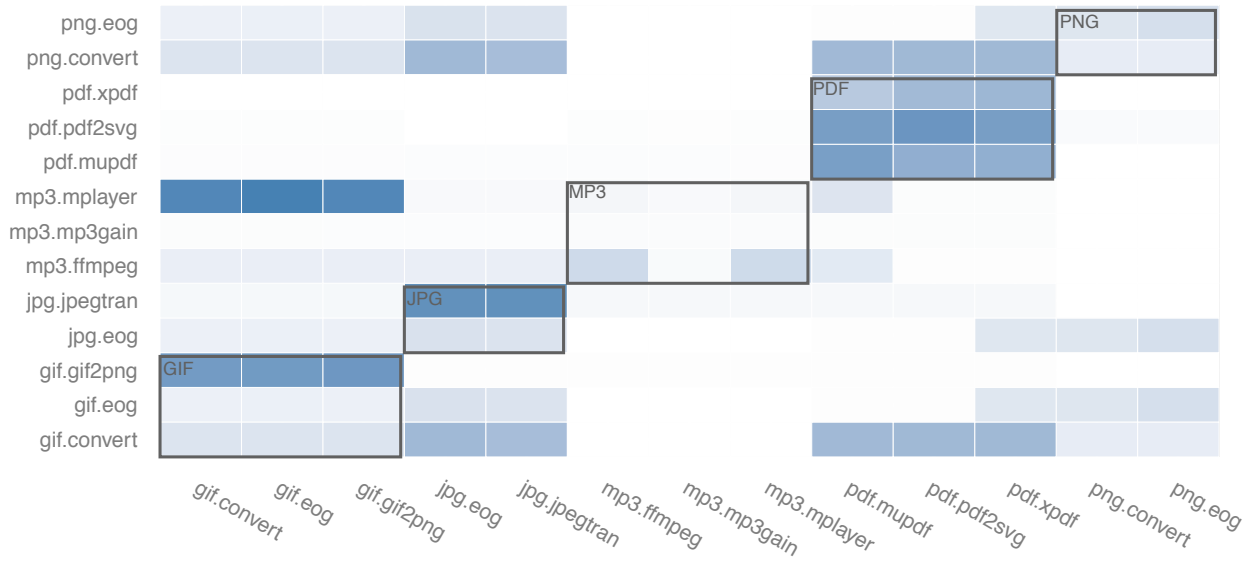


Figure 3.5: Transferability of UNWEIGHTED MINSET coverage across configurations. The base configurations, on which the reduced sets were computed, are on the bottom; the tested configurations are on the left. Darker colors indicate higher coverage.

surprising since `convert` can also process PDF files. Similar patterns can be similarly explained—for example, GIF minsets are performing *better* than MP3 minsets for `mplayer`, simply because `mplayer` can render GIF images.

The heat map allows us to see two clear patterns:

1. High coverage indicates the application accepts a file type. For instance, by following the row of the `gif.eog` configuration we can immediately see that `eog` accepts GIF, JPG, and PNG files, while it does not process MP3s or PDFs.

2. Coverage transfers across applications that process the same file type. For example, we clearly see the PDF cluster forming across all PDF configurations, despite differences in implementations. While `xpdf` and `pdf2svg` both use the `poppler` library for processing PDFs, `mupdf` has a completely independent implementation. Nevertheless, `mupdf`'s minset performs well on `xpdf` and vice versa. Our data shows that similar clusters appear throughout configurations of the same file type, suggesting that we can reuse minsets across applications that accept the same file type.

Conclusion: Reduced sets are transferable. Our data suggests that reduced sets can be transferred to programs parsing the same file types with respect to code coverage. Therefore, it is necessary to compute only one reduced set per file type.

3.6 Discussion

The proposed seed reduction algorithms were designed based on the fact that seeds with higher code coverage will find more number of bugs since we can test diverse parts of the code. What if an attacker tries to find bugs that can be obtained from seeds that cover fewer lines of code instead? Can an attacker take advantage by knowing the fact the defenders are using our seed reduction technique?

We believe that one can find a distinct set of bugs than the bugs that our technique would find by selecting seeds that cover the fewest lines of code. However, this cannot be a significant threat in practice, because the selected seeds will essentially cover a small portion of the code, and thus, one can only find bugs in the exercised code region.

3.7 Summary

In this chapter we designed and evaluated six seed selection techniques based on the observations on several bug characteristics. In addition, we formulated the optimal ex post facto seed selection scheduling problem as an integer linear programming problem to measure the quality of seed selection algorithms. We performed over 650 days worth of fuzzing to determine ground truth values and evaluated each algorithm. We found 240 new bugs. Our results suggest how best to use seed selection algorithms to maximize the number of bugs found.

Chapter 4

Parameter Inference

> If you realize that all things change, there is nothing you will try to hold on to. If you are not afraid of dying, there is nothing you cannot achieve.
>
> —Lao Tzu

Some fuzzing parameters such as mutation ratio can have arbitrary many values. Parameter reduction techniques introduced in Chapter 3 requires complete evaluation of all potential parameters, i.e., we need to execute the SUT once for each parameter value. Furthermore, there are millions of different executions to consider for a single mutation ratio. A natural question follows: can we directly infer a good parameter value without evaluating all candidates?

Current mutational fuzzers either manually select a single mutation ratio, or use random ratios from a range. Thus, the fundamental challenge is that they all involve either manual or non-adaptive parameter selection. First, an analyst has to choose the fuzzing parameters based on their expertise. For example, `zzuf` [98] runs with either a single or a range of mutation ratios, but the analyst must specify those parameters. Second, if not manual, the parameters are derived non-adaptively regardless of the program under test. BFF [86], for instance, splits a set of all possible nonzero mutation ratios into a predefined set of intervals, and performs scheduling (FCS) over the intervals. `zzuf` uses a predefined mutation ratio if a user does not specify a value. AFL (American Fuzzy Lop) [156] also employs several bit-flipping mutation strategies that only mutate a fixed number of bits, e.g., flip only a single random bit, regardless of the program under test.

In this chapter, we introduce our system, SymFuzz that automatically infers a good mutation

ratio—a parameter used to confine the Hamming distance from the seed to generated test cases for MBF—from a single program execution.

4.1 Exploiting Characteristics of Fuzzing Outcome

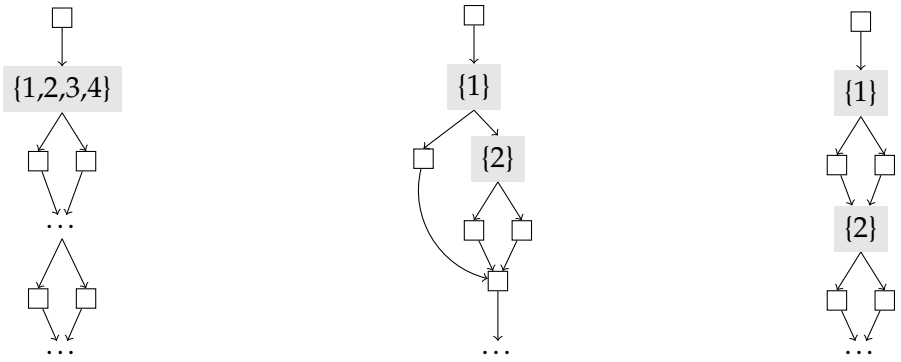
We observed that one can produce multiple crashing inputs by mutating a single seed, and if the crashing inputs are due to the same bug, they present similar patterns in their mutated bit positions. We use this characteristic to recognize a good mutation ratio for fuzzing. The primary intuition of our work is that a desirable mutation ratio that maximizes the fuzzing efficiency can be deduced from the dependence relations between the input bits of a seed for a program.

Suppose we are given a program and a 96-bit seed that consists of a 32-bit magic number followed by two consecutive 32-bit integer fields. We also assume that the magic number is 42424242_{16} and two integer values are zero. The program crashes when an input value satisfies the following two conditions: (1) the magic number remains 42424242_{16}; and (2) the third field is negative integer. To trigger the crash, one needs to flip the most significant bit in the third field, but never touch the bits in the first field of the seed. The value of the second field does not affect the crash. In this case, there exists a dependence relation between the first and the third field: the third field depends on the first field. That is, even though we have a negative value for the third field, we will never be able to trigger the crash if we flip any of the bits in the first field. In this chapter, we show that the dependence between the input bits indeed decides the best mutation ratio for this crash, which is about 0.031.

4.2 Input-Bit Dependence

We define dependence between input bits using control dependence [9]. Informally, when a node u in a Control-Flow Graph (CFG) decides if another node v is executed or not, then we say v is control-dependent on u. We extend the notion of control dependence to define the relationship between input bits as follows.

Definition 6 (*Input-Bit Dependence*)**.** Given an execution $\sigma_p(s)$, consider two bit positions x and y of s. We say that the bit s_x is *dependent* on s_y, denoted by $s_x \xrightarrow{dep(p)} s_y$, if either of the following



(a) Bits 1, 2, 3, 4 are dependent each other. (b) Bit 2 is dependent on bit 1. (c) Bits 1 and 2 are independent.

Figure 4.1: Input-bit dependence. Each gray box represents a conditional branch that is controlled by a set of input bit positions.

conditions holds: (1) there is a conditional branch that reads both s_x and s_y; (2) there are two conditional branches c_1 and c_2 that read s_x and s_y respectively, and c_1 is control-dependent on c_2.

Figure 4.1 demonstrates input-bit dependence for three different cases. Figure 4.1a and Figure 4.1b show examples that satisfy the first and the second condition of input-bit dependence respectively. In Figure 4.1a, every bit involved in the same condition is dependent on each other due to the first condition of Definition 6: $s_1 \xrightarrow{dep(p)} s_1, s_1 \xrightarrow{dep(p)} s_2, s_1 \xrightarrow{dep(p)} s_3, s_1 \xrightarrow{dep(p)} s_4, s_2 \xrightarrow{dep(p)} s_1, \cdots, s_4 \xrightarrow{dep(p)} s_3, s_4 \xrightarrow{dep(p)} s_4$. In Figure 4.1b, $s_2 \xrightarrow{dep(p)} s_1$. Finally, Figure 4.1c presents a case where two input bits are not dependent on each other.

As an example, let us consider the following C program.

```
char x = input[0]; char y = input[1];
if ( x > 42 ) {
  if ( y > 42 ) {
    ... /* omitted */
```

Given a program execution that exercises Line 4, the second byte (bits 9 to 16) of the input is dependent on the first byte (bits 1 to 8), all the bits in the first byte are dependent upon each other, and all the bits in the second byte are dependent upon each other.

Based on the definition of the input-bit dependence, we can compute the set of dependency bits for each bit in a seed; we use the term "dependency" in the same vein to the term "library dependencies". We call such a set as a *dependency bitset*, and denote it with a function $\uparrow$. The upward arrow is intended to reflect the direction of the dependence relation in a CFG. For instance,

$\uparrow_s^p(\{3,4\})$ is the set of bits that are depended on either by s_3 or by s_4 in the execution $\sigma_p(s)$.

Definition 7 (*Dependency Bitset*)**.** Given a set of bit positions X of a seed s for a program p, the dependency bitset of X is defined as

$$\uparrow_s^p(X) = \left\{ y \;\middle|\; x \in X, s_x \xrightarrow{dep(p)} s_y \right\}.$$

Intuitively, we have defined input-bit dependence to reveal the approximate syntactic structure of an input. Most input structures consist of a series of input fields. For instance, a PNG file has a series of data chunks each of which consists of four input fields. Intuitively, every bit in an input field should depend on each other, because all the bits together decide the control-flow of the program. Indeed, a notion similar to input-bit dependence has been used in recovering the format of an input from a given program execution [102]. More precisely, bits in a dependency bitset can be a superset of a bits in an input field: input-bit dependence can involve multiple input fields. In this chapter, we use the input-bit dependence to infer the optimal mutation ratio for surface-based mutational fuzzing (§4.4.3).

4.3 Failure Rate based on Mutation Ratio

Recall from §2.6, we defined the failure rate of surface-based mutational fuzzing. In this chapter, we are interested in finding a mutation ratio r that maximizes the failure rate of surface-based mutational fuzzing. Therefore, we need to represent the failure rate in terms of r. To do so, we first categorize bit positions in a seed into several kinds, and approximate the failure rate in terms of mutation ratio and input-bit dependence. In the rest of the chapter we are going to use the term mutational fuzzing to mean surface-based mutational fuzzing for simplicity. See §4.8 for more discussion about the type of mutational fuzzing algorithms.

Given a program p and a seed s, suppose the program crashes when it is executed on a mutated s, i.e., there is an input among the K-neighbors of s that triggers the crash. Specifically, there is a set of bits in s that, when flipped, generates a buggy input for p. We call such a set a *buggy bitset* of s. There can be multiple buggy bitsets for a single bug.

Definition 8 (*Buggy Bitset*)**.** Given a program p and an N-bit seed s, a buggy bitset is a set of bit positions $B \subseteq \{1, 2, \ldots, N\}$ where $\textsc{Triage}(\pi, \sigma_p(\mu(s, B))) \neq \perp$.

Some of the bits in a buggy bitset may not need to be flipped to generate an input that triggers the bug. Among all subsets of a buggy bitset, there exists a combination of bits with a minimum cardinality while still producing a buggy input for the same bug. We call such a subset a *minimum buggy bitset*.[1] Notice that there may be multiple minimum buggy bitsets with the same size. Suppose there is an 8-bit seed, and flipping both the first and second bits of the seed leads a program p to crash. The buggy bitset is therefore $\{1, 2\}$, and $\textsc{Triage}(\pi, \sigma_p(\mu(s, \{1, 2\}))) \neq \perp$. Now, suppose flipping only the second bit of the seed produces a buggy input that leads to the same crash, i.e., $\textsc{Triage}(\pi, \sigma_p(\mu(s, \{1, 2\}))) = \textsc{Triage}(\pi, \sigma_p(\mu(s, \{2\})))$. Since we assume that a seed does not produce a program crash by itself, a minimum buggy bitset is $\{2\}$.

Definition 9 (*Minimum Buggy Bitset*)**.** Given a buggy bitset B for a program p and a seed s, a minimum buggy bitset B' of B is an element of the set

$$\underset{\{B^* \subseteq B \mid \textsc{Triage}(\pi, \sigma_p(\mu(s, B^*))) = \textsc{Triage}(\pi, \sigma_p(\mu(s, B)))\}}{\arg\min} |B^*| \,.$$

A minimum buggy bitset B' includes a set of bits that must be flipped to generate a buggy input. Any bit position other than B', i.e., any element of $\{1, 2, \ldots, N\} \setminus B'$, either (1) does not affect the crash regardless of its values; or (2) thwarts the crash when it is flipped. We let a set of bits that must *not* be flipped for triggering the crash as *fixed bitset*, and denote it with F. $F(B')$ is a set of bits that must not be flipped to trigger the corresponding crash of B'.

We now compute a failure rate for each minimum buggy bitset. For simplicity, let b be the cardinality of a minimum buggy bitset ($b = |B'|$), and let f be the cardinality of the $F(B')$ ($f = |F(B')|$). We also let r be the mutation ratio. The failure rate of mutational fuzzing for B' follows a multivariate hypergeometric distribution [24], where the population size is N and the number of

[1] Deriving a minimum buggy bitset is often called bug minimization [85].

draws is $(N \times r)$. Therefore, the failure rate of a minimum buggy bitset that has b elements is:

$$\theta_b = \frac{\binom{b}{b}\binom{N-f}{N\cdot r-b}}{\binom{N}{N\cdot r}} = \frac{\binom{N-f}{N\cdot r-b}}{\binom{N}{N\cdot r}}, \quad \text{when } N \cdot r \geq b \tag{4.1}$$

This formula can be explained as follows. Given an N-bit seed, the total number of possible inputs that mutational fuzzing can generate is $\binom{N}{N\cdot r}$. To generate a buggy input from a seed, we need to flip all the bits in the minimum buggy bitset (this is the $\binom{b}{b}$ term), while not flipping the bits in the dependency bitset of B' (this is the $\binom{N-f}{N\cdot r-b}$ term). Since $\binom{b}{b} = 1$, the term can be eliminated.

To find a fixed bitset, a set of bits that must *not* be flipped for triggering the crash, we overapproximate a set of bits that changes the program execution with respect to the bits in B', which is a dependency bitset of B' by definition. If any of the bits in $\uparrow_s^p(B')$ are flipped, it will change the execution of the program. Since all the bits in B' must be flipped, the other bits in $(\uparrow_s^p(B') \setminus B')$ must *not* be flipped to maintain the same execution path for the crash.[2]

The above argument can be intuitively explained by an example. A bug is typically triggered when one or more input fields have specific values, e.g., one needs to set an integer field to be greater than the size of a program buffer to trigger a buffer overflow. However, even though the integer field has a value greater than the buffer size, the program might take an execution path that does not even read the values. This happens when the program checks the value of another input field x before it reaches the buffer overflow, and jumps to another execution path. Therefore, the integer input field is dependent on x. This is the key intuition of approximating the failure rate of mutational fuzzing in terms of the input-bit dependence.

Using the idea of the input-bit dependence, we can approximate the failure rate of a minimum buggy bitset from Equation 4.1 by replacing f with the cardinality of the dependency bitset of B'. For simplicity, we use d to denote the cardinality ($d = |\uparrow_s^p(B')|$). Then, the failure rate of a minimum buggy bitset that has b elements is:

$$\theta_b = \frac{\binom{N-f}{N\cdot r-b}}{\binom{N}{N\cdot r}} \approx \frac{\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}}, \quad \text{when } N \cdot r \geq b \tag{4.2}$$

The failure rate is only meaningful when the number of flipped bits is not less than the size

[2] The dependency bitset of B' is an over-approximation of the immutable bit positions for the crash, because flipping some bits in $(\uparrow_s^p(B') \setminus B')$ may still trigger the same crash.

of B', i.e., $N \cdot r \geq b$. When $N \cdot r < b$, we simply cannot flip every bit in B'. By the definition of the minimum buggy bitset (Definition 9), one needs to flip all the bits in B' in order to generate a buggy input. Therefore, the failure rate is effectively 0 in this case.

4.4 Mutation Ratio Optimization

In this section we introduce a systematic way, called *mutation ratio optimization*, of deciding a set of mutation ratios for a given program and a seed using the formal definitions addressed in §4.2 and §4.3. Our technique automatically adapts to a given program-seed pair, and it enables efficient bug finding for mutational fuzzing.

4.4.1 Mutation Ratio Optimization Challenge

We first address *mutation ratio optimization challenge* as follows.

Definition 10 (*Mutation Ratio Optimization Challenge*)**.** Given a program p and an N-bit seed s, consider a crash that is identified by a minimum buggy bitset B', and let $b = |B'|$. The mutation ratio optimization challenge is to derive a mutation ratio r that maximizes the failure rate θ_b of p.

Notice the cardinality of a minimum buggy bitset (b) is not known unless we have found the corresponding bug. Moreover, we may have multiple optimal mutation ratios for different values of b. Therefore, several questions remain: How do we solve the mutation ratio optimization challenge? How do we compute the cardinality of the dependency bitsets (d) for a given program-seed pair? We address these questions in the following sections.

4.4.2 Solving for an Optimal Mutation Ratio

Recall in §4.3 we described the failure rate θ_b of mutational fuzzing with respect to three variables: the bit size of a seed (N), the cardinality of a minimum buggy bitset (b), and the cardinality of a dependency bitset of the minimum buggy bitset (d). One of the primary challenges is to find a mutation ratio $0 < r \leq 1$ that maximizes θ_b. When $d = b$, i.e., $B' = \uparrow_s^p(B')$, it is trivial to show that the maximum failure rate is achieved with $r = 1$: we simply let $d = b$ and $r = 1$ from Equation 4.2, and then the failure rate θ_b becomes always 1 regardless of the value of b. When $d = N$, there is no

bit position to flip other than the ones in $\uparrow_s^p(B')$, and, as a result, the only way to trigger the crash is to flip exactly b bit positions. That is, the optimal mutation ratio is b/N. When $b < d < N$, we solve the mutation ratio optimization problem by modeling it as a classic nonlinear programming problem (NLP) [23] as follows.

For N, b, and d, find r to

$$\begin{aligned} \textbf{maximize} \quad & \theta_b = \frac{\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}} \\ \textbf{subject to} \quad & (0 < r \leq 1) \\ \wedge \quad & (b < d < N) \\ \wedge \quad & (b \leq N \cdot r \leq N - d + b). \end{aligned}$$

The first constraint of the NLP is from the definition of mutation ratio: mutation ratio must be between zero and one. The second constraint ($b < d < N$) is to restrict the range of the d value. When $d = b$, the optimal mutation ratio is 1, and when $d = N$, the optimal mutation ratio becomes b/N as we discussed above. The third constraint ($b \leq N \cdot r \leq N - d$) is due to our problem definition: (1) we should flip more than the cardinality of a minimum buggy bitset in order to generate an input that trigger the bug ($b \leq N \cdot r$); (2) we should not flip any bits in ($\uparrow_s^p(B') \setminus B'$), hence the maximum number of bit flips is ($N - d + b$).

We now solve the above NLP to obtain an optimal mutation ratio for a given minimum buggy bitset. The solution to it is the optimal mutation r with respect to b, d and N. See Appendix A.1 for a complete proof.

Theorem 2 (Optimal Mutation Ratio)**.** *Given a minimum buggy bitset B' and the corresponding $\uparrow_s^p(B')$ for a program p and a seed s, let $b = |B'|$ and $d = |\uparrow_s^p(B')|$. The optimal mutation ratio r for finding the bug* TRIAGE$(\pi, \sigma_p(\mu(s, B')))$ *is*

$$r = \frac{b \times (N+1)}{d \times N} \text{ when } N \cdot r > b. \tag{4.3}$$

We find an optimal mutation ratio r that maximizes the failure rate as follows when $b < d < N$. First, when $b = N \cdot r$, we compute a failure rate θ_1 by letting $r = b/N$ from Equation (4.2). Next, we

obtain another mutation ratio r_2 for the case of $b < N \cdot r$ using Equation (4.3). We then compute a failure rate θ_2 for r_2 from Equation (4.2). Finally, we compare the two failure rates and return an optimal mutation ratio as follows: if θ_1 is greater than θ_2 then the optimal ratio is b/N; otherwise it is r_2. We note, when computing r_2, N is given from the seed, but b and d are unknown. We know neither buggy bitsets nor minimum buggy bitsets prior to fuzzing the program under test: our goal is to pre-compute the optimal mutation ratio before fuzzing. That is, we cannot know the corresponding minimum buggy bitset before hitting a bug. This problem suggests that we must find a way to estimate the value of b and d without the prior knowledge about the buggy bitsets.

4.4.3 Estimating r

Suppose there exist M unique crashes that can be produced by mutating an N-bit seed s for a program p. Since each crash can have its own distinct minimum buggy bitsets, the value of b and d may differ depending on the crash, and thus, each crash may have different optimal mutation ratios. Ideally, one may find a set of distinct mutation ratios for all M crashes, but knowing exact b and d for every unique crash is infeasible in practice.

To estimate effective mutation ratios in finding all M crashes, we use the averaged values of b and d. Although buggy bitsets are unknown in advance, we can still compute dependency bitsets of every bit in the seed: we can obtain all possible d values from the given program-seed pair, which will expose the trend of the input-bit dependence of the seed. We then use this information to estimate d. Let $\mathcal{P}(S)$ be the powerset of S. We then denote $\bar{d}_{\text{all}}$ as the average cardinality of every possible dependency bitsets for the program-seed pair:

$$\bar{d}_{\text{all}} = \frac{\sum_{x \in \mathcal{P}(\{1,2,\ldots,N\})} |\uparrow_s^p(x)|}{2^N}.$$

Recall from §4.2, the input-bit dependence indicates the overall input structure for a given program-seed pair: it reveals which chunk of the input bits together affects the control flow of the program. Therefore, the average input-bit dependence $\bar{d}_{\text{all}}$ is an approximate indicator that shows how many bits are dependent on each other for a given program-seed pair. When $\bar{d}_{\text{all}}$ is high, that means many input bits in the seed are dependent on each other, and thus, there are likely to be more input bits that should not be mutated to trigger the crashes. That is, a larger $\bar{d}_{\text{all}}$

corresponds to a smaller r and vice versa. This relationship between $\bar{d}_{\text{all}}$ and r is evident from the above NLP formulation. The optimal mutation ratio ranges from b/N (when $d = N$) to 1.0 (when $d = b$), and as we have smaller d, i.e., less dependence between input bits, we have a higher optimal mutation ratio.

Although $\bar{d}_{\text{all}}$ shows the trend of input-bit dependence, there are two remaining problems in using it. First, just averaging the cardinality of all possible dependency bitsets is not necessarily the best way to represent the trend, because the cardinality of minimum buggy bitsets (b) may be biased towards several values. Second, the number of dependency bitsets to consider is exponential in N, and N is typically not small.

To mitigate both challenges, we incorporate the distribution of b (β) into the average input-bit dependence by using adaptive sampling [143], which also helps in computing an approximate average efficiently. The algorithm is shown in Algorithm 4.1. First, we select a random cardinality b with the probability associated with each cardinality in β (Line 4, `WeightedRand`). Next, we sample a set of random bit positions S of cardinality b (Line 5, `RandomK`). `RandomK` takes in N and b, and returns b distinct random numbers from the interval $[1, N]$. We then compute $|\uparrow_s^p(S)|$, and use the cardinality to compute a new cardinality sum. Then, we check the difference between the previous and the new mean values to see if it is smaller than a threshold ϵ (Line 8). We repeat the process until the difference is negligible (we use $\epsilon = 10^{-7}$ in our experiment). After breaking out of the while loop, the algorithm returns the final average input-bit dependence denoted as $\bar{d}$.

Since $\bar{d}$ relies on the distribution of b (β), it is important to note how the distribution looks like. We obtained a large scale fuzzing dataset from a previous work, and computed the cardinality of minimum buggy bitsets for each unique crash found in [131]. The average b value was 9 and the standard deviation was 18. This result conforms to the observations from practitioners [85]. See §4.7.3 for further discussion on how we obtained b values from the dataset.

Now that we have a distribution of b values from a large-scale experiment, we need to estimate r using the averaged d value ($\bar{d}$). Since $\bar{d}$ is the average cardinality of dependency bitsets per each bit in minimum buggy bitsets, we can estimate the cardinality of a dependency bitset for a minimum buggy bitset of cardinality b using $b \times \bar{d}$. By letting $d = \bar{d} \times b$, we can simplify the Equation 4.3 as

Algorithm 4.1: Computing $\bar{d}$ using adaptive sampling.

input : A distribution of b values (β)
output: $\bar{d}$
prevN $\leftarrow 0$
prevSum $\leftarrow 0$
while *true* **do**
  $b \leftarrow$ `WeightedRand(`N, β`)`
  $S \leftarrow$ `RandomK(`N, b`)`
  newN $\leftarrow$ prevN $+$ b
  newSum $\leftarrow$ prevSum $+ |\uparrow_s^p(S)|$
  if prevN $\neq 0 \wedge |{}^{\text{newSum}}/_{\text{newN}} - {}^{\text{prevSum}}/_{\text{prevN}}| < \epsilon$ **then break**
  else prevN $\leftarrow$ newN; prevSum $\leftarrow$ newSum
end
return ${}^{\text{prevSum}}/_{\text{prevN}}$ `/* Returns` $\bar{d}$ `*/`

follows.

$$r = \frac{b \times (N+1)}{b \times \bar{d} \times N} = \frac{1}{\bar{d}} \cdot \frac{N+1}{N}. \tag{4.4}$$

The value of b is now included in $\bar{d}$, and we only need to consider the value of $\bar{d}$ to estimate the optimal mutation ratio r. Given the distribution of b in crashes, $\bar{d}$ provides a way to estimate the cardinality of dependency bitsets for the crashes, which, in turn, helps in estimating r.

4.5 Input-Bit Dependence Inference

At a high level, Input-Bit Dependence Inference (IBDI) is a process of computing the input-bit dependences for every bit in a seed from a program execution. We then use these dependence relations to compute $\bar{d}$ as in Algorithm 4.1. From the perspective of program analysis, IBDI is a symbolic analysis that is more specific than the traditional taint analysis [46, 124, 155], and more abstract than the traditional symbolic execution [26, 88, 94]. Our approach is inspired by several automatic input format recovery approaches including [31, 49, 54, 102], where they share a common theme as us: they use a program execution to reveal the structure of an input. However, our focus is on figuring out the input-bit dependence rather than precise input formats.

p	::=	stmt*
exp	::=	get_input(src) | load(exp) | var
		| exp $\Diamond_b$ exp | $\Diamond_u$ exp | v
stmt	::=	var:= exp
	|	goto exp
	|	if exp then goto exp_1 else goto exp_2
	|	call exp
	|	ret exp
	|	store(exp,exp)
v	::=	⟨unsigned integer, a set of affecting bits⟩
$\Diamond_b$	::=	binary operators
$\Diamond_u$	::=	unary operators

Table 4.1: A simple language for IBDI.

Context	Meaning
Σ	a list of program statements
μ_c	mapping from an address to concrete value
Δ_c	mapping from a variable to concrete value
μ_a	mapping from an address to abstract value
Δ_a	mapping from a variable to abstract value
Γ	current dependence predicate
c	current input-dependence stack
l	current delay queue
pc	current program counter
i	current statement

Table 4.2: The execution context of our analysis.

4.5.1 The Algorithm

Input-Bit Dependence Inference (IBDI) takes as input a program and a seed, and outputs the input-bit dependence for every bit of the seed. Similar to dynamic symbolic execution [34, 71], IBDI runs the program under test both concretely and symbolically. The key difference between IBDI and dynamic symbolic execution is that IBDI operates on a set of dependent bits instead of generating bit-vector-level path formulas, hence it does not rely on SMT solvers [55]. As in dynamic symbolic execution, IBDI introduces symbolic values whenever reading from a user input, e.g., `read` system call. It then symbolically evaluates program statements on a program execution. It also constructs a CFG while symbolically executing the program in order to compute control dependences between variables and the corresponding input bits.

We describe our IBDI algorithm using the formal runtime semantics over a simple language

shown in Table 4.1. In our language, a program is a sequence of statements. There are four different jump statements including `goto`, `if-else`, `call`, and `ret`. The first two are regular jump statements: `goto` is an unconditional jump statement, and `if-else` is a conditional jump statement. The last two kinds, `call` and `ret`, are special jump instructions that represent calls and returns respectively. Notice, however, we do not allow call/ret instructions to implicitly manipulate call-stacks in our language. For example, call instruction in x86 will be jitted into into stack manipulation statements followed by a `call` statement.

Since we execute a program both concretely and symbolically, expressions in our language evaluate to a value v, which is a tuple of a concrete value and an abstract value. A concrete value is an unsigned integer, and an abstract value is a set of input bits that affects either directly or indirectly the value, which is often called data lineage [102, 103]. We denote data lineage of a variable as a set of bit positions. For example, if a variable x evaluates to $\langle 1, \{2, 3, 4\}\rangle$, it means the variable x has a concrete value of 1, and is also affected by the three other input bits.

We use $\Diamond_b$ to denote binary operators such as addition, subtraction, etc. Similarly, $\Diamond_u$ represents unary operators such as minus. When we evaluate $\Diamond_b$ over abstract values (data lineages), we apply set union between them. For example, when we evaluate a subtraction between $\{1\}$ and $\{1, 2, 3, 4\}$, we obtain $\{1, 2, 3, 4\}$. For $\Diamond_u$, we simply propagate abstract values from a source to a destination.

The execution context of our analysis consists of ten fields as shown in Table 4.2. We store abstract and concrete values for variables in Δ_a and Δ_c respectively in a map.[3] Similarly, we store abstract and concrete values of memory addresses in μ_a and μ_c respectively in a map. To access maps, we use a bracket notation. For example, $\Delta_a[x]$ returns the current abstract value of x, and $\Delta_c[x \leftarrow 1]$ returns a new map, which is equivalent to the previous map except that the value of x is 1. We use $\Downarrow$ to represent evaluation of an expression under a given context. For example, $\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow v$ is an evaluation of an expression e to a value v in the context given as 4-tuples $(\mu_c, \Delta_c, \mu_a, \Delta_a)$.

We encode the input-bit dependence for every bit in an input using a data structure that we call *dependence predicate* (Γ). The dependence predicate is essentially a map from a bit of an input to a set of bit positions that the bit is dependent on. As we execute the program under test, we update Γ using a `merge` function. For example, suppose Γ has a mapping from the first bit to $\{3, 4\}$. Then,

[3] Variables at the machine-code level are really *r*egisters.

$$\frac{v_c = \text{a concrete value from } src \quad v_a = \{\text{set of input bit positions}\}}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash \texttt{get\_input(src)} \Downarrow \langle v_c, v_a \rangle} \textsc{Input}$$

$$\frac{}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash \texttt{var} \Downarrow \langle \Delta_c[\texttt{var}], \Delta_a[\texttt{var}] \rangle} \textsc{Var}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad v'_c = \mu_c[v_c] \quad v'_a = \mu_a[v_c] \Diamond_b v_a}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash \text{load } e \Downarrow \langle v'_c, v'_a \rangle} \textsc{Load}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad v'_c = \Diamond_u v_c}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash \Diamond_u e \Downarrow \langle v'_c, v_a \rangle} \textsc{Unary-Op}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1} \rangle \quad \mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2} \rangle}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_1 \Diamond_b e_2 \Downarrow \langle v_{c1} \Diamond_b v_{c2}, v_{a1} \Diamond_b v_{a2} \rangle} \textsc{Binary-Op}$$

$$\frac{}{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash v \Downarrow \langle v, \{\} \rangle} \textsc{Const}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad \begin{array}{l}\Delta'_c = \Delta_c[var \leftarrow v_c] \\ \Delta'_a = \Delta_a[var \leftarrow v_a]\end{array} \quad \begin{array}{l}c' = \text{checkIDS}(c, pc) \\ \boxed{l' = l.\text{add}(\langle v_a, c'.\text{top}() \rangle)}\end{array} \quad i = \Sigma[pc+1]}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \texttt{var} := e \rightsquigarrow \Sigma, \mu_c, \Delta'_c, \mu_a, \Delta'_a, \Gamma, c', l', pc+1, i} \textsc{Assign}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad c' = \text{checkIDS}(c, pc) \quad i = \Sigma[v_c]}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{goto } e \rightsquigarrow \Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c', l, v_c, i} \textsc{Goto}$$

$$\frac{\begin{array}{lll}\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle 1, v_a \rangle & c = \text{checkIDS}(c, pc) & \\ \mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1} \rangle & c' = \text{updateIDS}(c, pc, v_a) & l' = l.\text{add}(\langle v_{a1} \cup v_a, c'.\text{top}() \rangle) \quad i = \Sigma[v_{c1}]\end{array}}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c', l', v_{c1}, i} \textsc{True-Cond}$$

$$\frac{\begin{array}{lll}\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle 0, v_a \rangle & c = \text{checkIDS}(c, pc) & \\ \mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2} \rangle & c' = \text{updateIDS}(c, pc, v_a) & l' = l.\text{add}(\langle v_{a2} \cup v_a, c'.\text{top}() \rangle) \quad i = \Sigma[v_{c2}]\end{array}}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c', l', v_{c2}, i} \textsc{False-Cond}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad c = \text{checkIDS}(c, pc) \quad c' = c.\text{push}(\langle pc+1, c.\text{pop}() \rangle) \quad i = \Sigma[v_c]}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{call } e \rightsquigarrow \Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c', l, v_c, i} \textsc{Call}$$

$$\frac{\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e \Downarrow \langle v_c, v_a \rangle \quad c' = \text{returnIDS}(c, pc) \quad \langle \Gamma', l' \rangle = \text{delayedUpdate}(\Gamma, l) \quad i = \Sigma[v_c]}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{ret } e \rightsquigarrow \Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma', c', l', v_c, i} \textsc{Ret}$$

$$\frac{\begin{array}{lll}\mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1} \rangle & c' = \text{checkIDS}(c, pc) & \mu'_c = \mu_c[v_{c1} \leftarrow v_{c2}] \quad i = \Sigma[pc+1] \\ \mu_c, \Delta_c, \mu_a, \Delta_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2} \rangle & & \mu'_a = \mu_a[v_{c1} \leftarrow v_{a1} \Diamond_b v_{a2}]\end{array}}{\Sigma, \mu_c, \Delta_c, \mu_a, \Delta_a, \Gamma, c, l, pc, \text{store}(e_1, e_2) \rightsquigarrow \Sigma, \mu'_c, \Delta_c, \mu'_a, \Delta_a, \Gamma, c', l, pc+1, i} \textsc{Store}$$

Figure 4.2: Operational semantics of input-bit dependence inference.

Algorithm 4.2: Dependence predicate update algorithm.

```
Function merge(Γ, X, Y)
    Γ' ← Γ
    for x ∈ X do
        Γ' ← Γ' [x ← (Γ[x] ∪ Y \ {x})]
    end
    return Γ'
end
```

Algorithm 4.3: Delay queue update algorithm.

```
Function delayedUpdate(Γ, l)
    for ⟨X, Y⟩ ∈ l do
        if ⟨X, Y⟩ is not memoized then
            Γ ← merge(Γ, X, Y)
            memoize ⟨X, Y⟩
        end
    end
    return ⟨Γ, [ ]⟩
end
```

merge($\Gamma, \{1, 2\}, \{5, 6\}$) will return a new dependence predicate which contains two mappings: (1) from the first bit to $\{3, 4, 5, 6\}$, and (2) from the second bit to $\{5, 6\}$. Algorithm 4.2 describes the merge function. Notice, in Line 4 of the algorithm, we compute the relative complement of $\{x\}$ in order to exclude the dependence relations that self-referencing.

One may call the merge function for every instruction encountered on the fly. However, we delay the predicate update until we reach a return instruction—thus, update the dependence predicate per control-dependent region [152]—for two reasons. First, it is more cache-efficient to perform updates once in a while. Second, we can employ a heuristic to eliminate unnecessary updates: there are many duplicated updates, thus we can memoize the last updates to speed up the process. To perform delayed update, we employ an additional field in our execution context, which we call a delay queue (l). l stores a tuple of the current data lineage and the current set of dependent bits from the control stack. To add an entry to a delay queue, we use an `add` method.

We maintain an input-bit dependence stack (c) to store a set of bit positions that the current instruction is control-dependent on. The idea is similar to dynamic control-dependence analysis in [152], but input-bit dependence stack maintains a set of control-dependent bits instead of storing

control-dependent statements. We use three methods to access the input-dependence stack (IDS): `top`() returns the top element of a stack, `pop`() returns a tuple of the top element of a stack and a new stack without the top element, and `push`(X) returns a new stack which contains an additional element X.

Each element on the stack is a 2-tuple (an address of an instruction that is beyond the scope of the current control dependence, a set of control-dependent bits). In our analysis, control dependence of a conditional branch is valid in two scenarios: (1) until we reach an immediate post-dominator of the conditional branch; (2) until we reach a return instruction. Therefore, we need to update the input-bit dependence stack either when we enter a function (call instructions), or when we encounter a conditional branch. However, to efficiently handle recursions, we use the same intuition as [152]: either when we enter the same function more than once or when we have the same immediate post-dominator, we replace the top element of the stack instead of pushing a new one.

Algorithm 4.4 illustrates three major functions to access the IDS. For every conditional branch, we call `updateIDS` to register a control-dependent region [152] from a conditional branch to an immediate post-dominator. For every return statement, we call `returnIDS` to clear up the IDS. We also update the IDS for every call instruction. Finally, we call `checkIDS` for every statement to check whether we have encountered the end of control-dependent region. When a CFG is incomplete, i.e., indirect jumps, we might not be able to find an immediate post-dominator. In this case, we use a conservative approach: we always merge the current set of control-dependent bits with the top element of the IDS.

We formulate the algorithm of input-bit dependence inference in an operational semantics in Figure 4.2. In the rule of assignment statement, we highlight the delay queue update using a box, because we optionally disable the function. In fact, even though we do not update the delay queue in assignment statements, we can still capture most of the input-bit dependence conditional branches. If we do not take the input-bit dependence for assignment statements, we may miss some dependence due to implicit data flow [93].

Algorithm 4.4: Input-dependence stack update algorithm.

Function `updateIDS`(c, pc, v_a)
  $pd \leftarrow$ immediate post-dominator of the current instruction at pc
  $\langle toppd, topdep \rangle \leftarrow c$.top()
  if $toppd = pd$ **then**
    $\langle \cdot, c \rangle \leftarrow c$.pop()
  end
  $c' \leftarrow c$.push$(\langle pd, v_a \cup topdep \rangle)$
  return c'
end
Function `returnIDS`(c, pc)
  $\langle toppd, \cdot \rangle \leftarrow c$.top()
  while c.*isNotEmpty*$() \wedge toppd \neq pc$ **do**
    $\langle \cdot, c \rangle \leftarrow c$.pop()
  end
  if c.*isNotEmpty*$()$ **then**
    $\langle \cdot, c \rangle \leftarrow c$.pop()
  end
  return c
end
Function `checkIDS`(c, pc)
  $\langle toppd, \cdot \rangle \leftarrow c$.top()
  if c.*isNotEmpty*$() \wedge toppd = pc$ **then**
    $\langle \cdot, c \rangle \leftarrow c$.pop()
  end
  return c
end

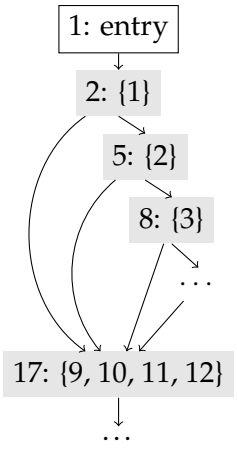
4.5.2 Example

Figure 4.3 is our running example showing a typical PNG parsing algorithm. It parses the first 8 characters using a series of conditional branches—which is an unrolled version of a for loop—from Line 2 to 14. It then reads the next 4 bytes as an integer in Line 16, and checks the value in Line 17. Figure 4.3b shows a control flow of the program, where each node is annotated with a line number of a branch instruction and a set of input bits affecting the condition of the branch at runtime. We use C to describe IBDI algorithm, but our system runs on raw binary executables. Additionally, we represent input positions in a byte-level granularity in our example for brevity.

Suppose we provide a valid PNG file to the parser, and follow the execution path of 1, 2, 5, 8, 14,

```
/* input read in buf */
if (buf[0] != '\x89'){
  error();
}else{
  if (buf[1] != '\x50'){
    error();
  }else{
    if (buf[2] != '\x4e'){
      error();
    }else{
      ...
    }
  }
}
/* next field */
len = *((*int32)&buf[8]);
if (len > PNG_MAX)
  error();
...
```

(a) A PNG parser in C



(b) A control-flow graph.

Line	Γ	c	l
1	$\cdot$	$\cdot$	$\cdot$
2	$\cdot$	$\langle 14, \{1\}\rangle$	$\langle\{1\}, \{1\}\rangle$
5	$\cdot$	$\langle 14, \{1, 2\}\rangle$	$\langle\{1\}, \{1\}\rangle$; $\langle\{2\}, \{1, 2\}\rangle$
8	$\cdot$	$\langle 14, \{1, 2, 3\}\rangle$	$\langle\{1\}, \{1\}\rangle$; $\langle\{2\}, \{1, 2\}\rangle$; $\langle\{3\}, \{1, 2, 3\}\rangle$
14	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ...	$\cdot$	$\cdot$
17	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ...	$\langle 19, \{9, 10, 11, 12\}\rangle$	$\langle\{9, 10, 11, 12\}$, $\{9, 10, 11, 12\}\rangle$
19	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ... $9 \mapsto \{9, 10, 11, 12\}$ $10 \mapsto \{9, 10, 11, 12\}$ ...	$\cdot$	$\cdot$

(c) The state transition table where each row is the execution context after executing the corresponding line. For delay queue l, each item is separated with a semicolon. The second column contains a mapping from a byte position to a set of byte positions.

Figure 4.3: A PNG parser example. We represent the input positions using a *byte*-level granularity in this figure for brevity.

16, 17, and 19. On Line 1, Γ, l, and c are empty. When the first conditional branch is encountered on Line 2, we check which input bytes are affecting the condition. Since the first byte is affecting the condition, we call `updateIDS`$(c, 2, \{1\})$, which first statically expands the control flow from the instruction on Line 2, and computes the immediate post-dominator of the instruction, which is Line 14 in this case. Then it updates c to have an element of $\langle 14, \{1\}\rangle$. Next, we push the input byte information $\langle\{1\}, \{1\}\rangle$ into l, which represent a dependence relation from the first byte to the first byte itself.[4]

On Line 5, we reach another conditional branch, which has a condition affected by the second byte. Since the top element of c has the same address as the immediate post-dominator of the branch, we replace the top element of c with $\langle 14, \{1, 2\}\rangle$ (due to Line 4-7 of `updateIDS`). The delayed queue is also updated with the updated control-dependence, which will call `merge`$(\Gamma, \{2\}, \{1, 2\})$ later in the `delayedUpdate` function. Similarly, we update the delay queue and the IDS until we reach the Line 14. Since the current instruction has the same address as in the top element of the IDS, we pop one element from the IDS (Line 23 of Algorithm 4.4), and then we call `delayedUpdate` of Algorithm 4.3 to update Γ. After executing Line 14, Γ has a mapping from each byte to the byte positions that the byte is dependent on. To be more precise, Γ should represent bit-level dependences, but we show byte-level dependences for simplicity. We perform the similar steps along the execution.

4.6 SymFuzz Design

In this section, we describe SymFuzz [39], a system that automatically finds an optimal mutation ratio for mutational fuzzing based on the input-bit dependence inference. Figure 4.4 summarizes our system design, which consists of two major components: symbolic analysis and mutational fuzzing. The symbolic analysis module takes in a program and a seed, and returns a recommended optimal mutation ratio. The mutational fuzzing module then uses the mutation ratio to perform fuzzing, and outputs buggy inputs found. Finally, we triage buggy inputs using our safe stack hash technique described in §4.6.3.

[4] In a bit-level granularity, this represents the input-bit dependences between the first eight input bits, where each of the bits is dependent on each other.

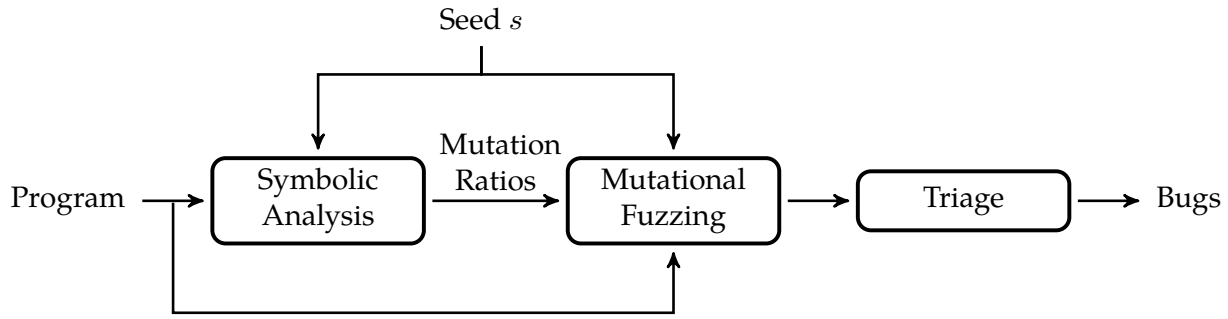


Figure 4.4: SymFuzz architecture.

4.6.1 Implementation

We base our system for input-bit dependence inference on BAP [28], an open-source binary analysis framework. BAP converts an x86 executable to an intermediate language suitable for program analysis. SymFuzz consists of 5,300 lines of OCaml for symbolic analysis and 1,600 lines of C++ code for instrumentation. We leverage PIN [104] to instrument a target binary. We also implement our mutational fuzzing framework in about 1900 lines of OCaml and 700 lines of C++.

4.6.2 Symbolic Analysis

The symbolic analysis module implements the operational semantics described in Figure 4.2 on top of the BAP [28] intermediate language. We employ several optimizations to our analysis including (1) tainted-block optimization, (2) JIT and PD caching, and (3) set memoization.

First, we use the taint information of each basic block to reduce the cost of symbolic analysis as follows. For each instrumented basic block, we perform a lightweight taint analysis. When a basic block does not involve any tainted instructions, we do not perform the symbolic analysis, and proceed to the next block. Notice our symbolic analysis inherently provides precise taint information for each block. Therefore, we do not need to maintain additional data structure for storing taints. In fact, the data-lineage tracking [102, 103] part of our symbolic analysis is more specific than the traditional taint analysis [46, 124, 155], and more abstract than the traditional symbolic execution [26, 88, 94].

Second, we employ several caches to improve the performance. The JIT cache is to cache recently-used BAP ILs, and the PD cache is to store the recently-computed immediate post-dominator

nodes. We note that expanding a static CFG for each conditional branch to compute an immediate post-dominator is an expensive operation compared to symbolic evaluation. This is because it involves not only jitting but also recursive disassembling and graph analysis.

Finally, we note that IBDI uses significant amount of memory footprint, because each bit in a seed needs to store a pointer to a set of bit positions, and each memory bit that is touched by the program under test also stores such a set for in μ_a. Although we perform byte-level analysis in our implementation, this problem still remains. The crux of the problem is that there can be multiple instances of the same set representation. Therefore, we memoize every set throughout the analysis, and make sure that there exists physically a single distinct set throughout the analysis.

4.6.3 Safe Stack Hash

Security practitioners use a call-stack trace or a part of a call-stack trace [118], e.g., taking only top five entries of a full stack-trace as in the fuzzy stack hash [122]. The rationale is that if two crashes have the same call-stack traces, then they are likely to have an equivalent final program state, and thus, it is an evidence of having the same root cause. This approach works for many cases, but it exhibits a false bucketing problem: it can put a single bug into multiple buckets.

We note that this false bucketing problem can significantly increase the number of bugs found for fuzzing especially when a buffer overflow mangles the return addresses on the stack. For example, suppose a mutated input data overwrites a return address of a call stack. The return address of the stack trace may contain any arbitrary values due to the mutation. In the worst case, we can have 2^{32} distinct call-stack traces on 32-bit machine just because of the mangled return address.

To mitigate this problem, we employ a technique, called *safe stack hash*, which stops traversing the call stack when it finds an unreachable return address. Specifically, we check for each return address of a call-stack trace starting from the top, i.e., the crashing stack frame, whether each return address falls in a mapped page. If not, we assume that the stack is mangled in the corresponding stack frame, and discard the rest of the return addresses in the call-stack trace. We also use the same heuristic as the fuzzy stack hash, and consider only the top five stack entries when computing the hash. Notice the number of bugs found from safe stack hash can only be less than the one from regular stack hash techniques such as the fuzzy stack hash. We implemented our safe stack hash using a GDB script written in Python.

Program	#Crashes	#Bugs	Seed Size (bits)	Seed Type
abcm2ps	299,204	34	35,040	abc
autotrace	15,848	23	16,304	bmp
bib2xml	603	2	177,152	bib
catdvi	2,045,327	8	1,632	dvi
figtoipe	443,301	38	8,016	fig
gif2png	21,600	3	1,816	gif
pdf2svg	125	1	23,368	pdf
mupdf	30	5	23,368	pdf
Total	2,826,038	114		

Table 4.3: The ground truth data.

4.7 Evaluation

We now evaluate our system SymFuzz on 8 real-world applications in order to answer the following questions.

1. Does it make sense to optimize the mutation ratio in mutational fuzzing? How does the number of bugs differ per mutation ratio? (§4.7.2)

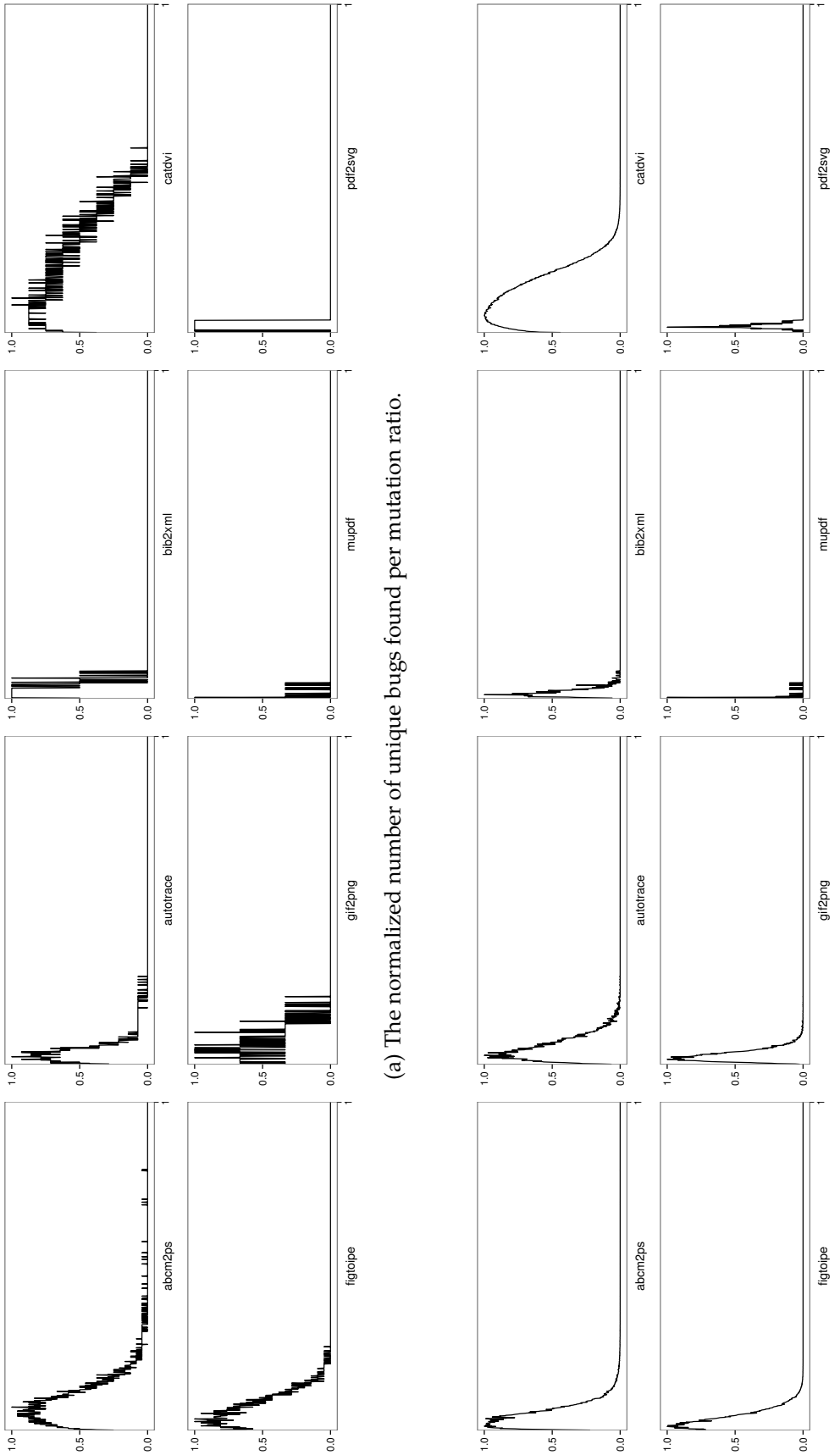
2. What is the cardinality of minimum buggy bitsets? Is the conventional wisdom about choosing small mutation ratios correct? (§4.7.3)

3. How effective is it to use the SymFuzz's adaptive strategy in terms of number of bugs found? (§4.7.4)

4. Does SymFuzz work well in practice? How many bugs did we find compared to the practical fuzzers such as BFF, `zzuf`, AFL? (§4.7.5)

4.7.1 Experimental Setup

We ran experiments on a private cluster consisting of 8 virtual machines. Each VM was running Debian Linux 7.4 on a single Intel 2.68 GHz Xeon core with 1GB of RAM, and all the applications that we tested were up-to-date as of May 2014. Each VM in our cluster was committed to only a single application throughout the experiments. The number of bugs reported from this chapter is based on our safe stack hash introduced in §4.6.3.



(a) The normalized number of unique bugs found per mutation ratio.

(b) The normalized number of crashes found per mutation ratio.

Figure 4.5: The effectiveness of fuzzing per mutation ratio evaluated over 1,000 mutation ratios from 0.001 to 1.000.

Program	Version	Size (bytes)
abcm2ps	6.6.17-1	347,664
autotrace	0.31.1-16	20,464
bib2xml	4.12-5	10,564
catdvi	0.14-12.1	93,220
figtoipe	20080517-1	48,644
gif2png	2.5.8-1	27,512
mupdf	0.9-2	5,999,472
pdf2svg	0.2.1-2+b3	6,548

Table 4.4: Applications used to compare ball-based and surface-based mutational fuzzing.

Collecting Ground Truth. We ran the mutational fuzzing module of SymFuzz individually to gather the ground-truth data of mutational fuzzing. We initially obtained a list of 100 file-conversion applications of Debian as in Chapter 5, and manually created a seed file for each application. We then fuzzed all 100 program-seed pairs with BFF [86] to know which programs exhibit crashes. We found 8 programs that resulted in at least one crash (see Table 4.4). We first ran our tool on each of the programs for 1,000 hours using 1,000 distinct mutation ratios from 0.001 to 1.000, i.e., 1 hour per each mutation ratio. Table 4.3 summarizes our ground truth experiment. In total, we have spent 8,000 CPU hours fuzzing the applications, and found 114 previously unknown bugs based on our safe stack hash. Since all the applications that we tested read in an input file, all the bugs found are potentially on the attack surface. For example, an attacker can craft a malicious file and upload it to the Internet, or send it as an email attachment in order to compromise users that run the applications with the file. We leave it as future work to check the exploitability of the bugs found [15, 38, 65].

4.7.2 Mutation Ratio Optimization

To justify our research, we first studied our ground truth data from fuzzing, and measured how the effectiveness of fuzzing changes with respect to the mutation ratio. We answer two specific questions as follows. First, is it meaningful to optimize the mutation ratio? Second, what is the potential benefit of using an adaptive optimization for fuzzing?

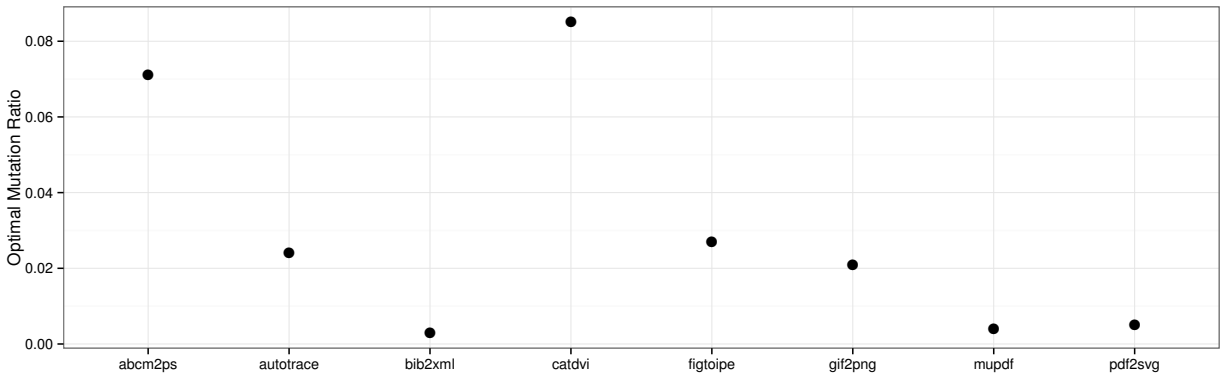


Figure 4.6: Empirically best mutation ratios for 8 programs.

Is Mutation Ratio Optimization Useful?

Optimizing mutation ratio is useful when the result of fuzzing varies significantly depending on a mutation ratio, and when there is a clear bias in the distribution of mutation ratios in terms of fuzzing efficiency. Figure 4.5a and Figure 4.5b illustrate respectively the normalized number of bugs and crashes found for each of the 8 programs in our ground truth dataset, that is, the number of bugs (crashes) divided by the maximum attainable number of bugs (crahses). Both figures show that the effectiveness of fuzzing largely depends on the mutation ratio. For example, we found the maximum number of bugs from `abcm2ps` using the mutation ratio of 0.071, but did not find any bug from the same program using the mutation ratio of 0.262. However, using the mutation ratio of 0.071 on `pdf2svg`, we found no bug in our dataset.

We note that the optimal mutation ratios differ across the programs. Figure 4.6 shows an empirically optimal mutation ratio per program based on the number of bugs found. The optimal ratios range from 0.003 to 0.085 depending on the program under test. We also notice fuzzing efficiency is biased towards the optimal mutation ratios from both the figures. Thus, our data suggest that mutation ratio optimization is useful in fuzzing.

How Much Better to Use Adaptive Optimization?

An immediate follow-up question of the first question is: how much better can adaptive optimization strategies be compared to non-adaptive strategies? In particular, we want to know what is the potential gain of using an adaptive strategy over non-adaptive strategies such as selecting ei-

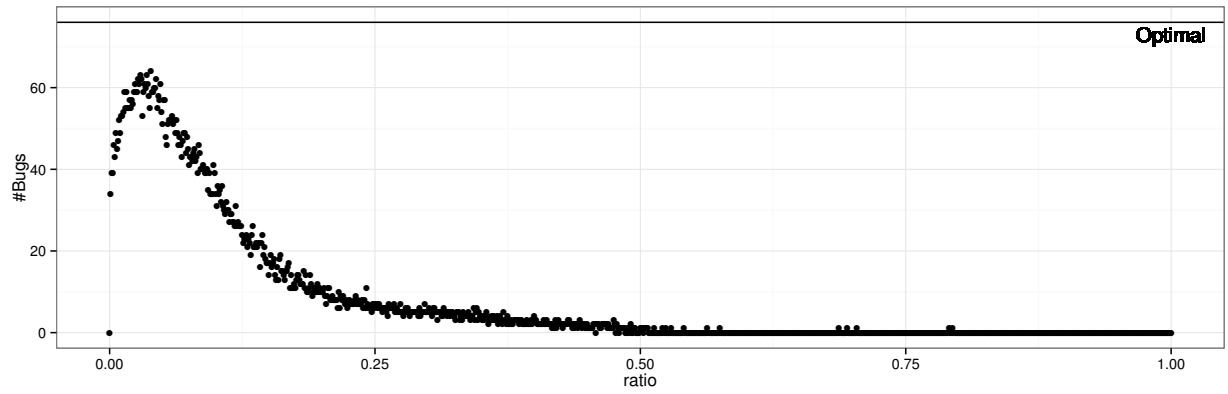


Figure 4.7: Comparison between a non-adaptive method, which is to choose a single default mutation ratio, and an adaptive (optimal) method, which is to select an empirically optimal mutation ratio per program.

ther (1) a single default mutation ratio, or (2) multiple ratios at random from a given range. Both the approaches are indeed employed in `zzuf` [98]. To answer the question, we first computed the maximum possible number of bugs that can be obtained by an optimal adaptive strategy for each program from our dataset; it was 76. We then compared this number against the number of bugs that can be found from the non-adaptive strategies.

The first non-adaptive strategy we checked is to choose a single default mutation ratio throughout an entire fuzzing campaign. Figure 4.7 shows the comparison. For all the mutation ratios in our dataset, the optimal adaptive strategy—represented as the horizontal line at the top of the figure—always found more bugs than the non-adaptive way. Moreover, even for the best case of the non-adaptive strategy, which is to choose the ratio of 0.039, the adaptive optimization found 18.8% more bugs compared to the non-adaptive method. Additionally, we notice that if we consider only a single mutation ratio per program, even a perfect adaptive strategy can only find 76 bugs out of 114 from our dataset. This result suggests the need for inferring multiple instead of a single mutation ratio, although this is outside the scope of this dissertation (see §4.8 for discussion).

The second strategy that we evaluated is to select a fixed range of mutation ratios throughout a fuzzing campaign. We used three different ranges suggested by the `zzuf` manual for this comparison, namely, $[0.00001, 0.01]$, $[0.00001, 0.02]$, and $[0.00001, 0.10]$. We fuzzed each application in our dataset for 1 hour with each of the ranges. In this experiment, we used the same algorithm that `zzuf` employs for selecting mutation ratios from a given range, which works as follows. We first

discretize the given range into a set of uniformly distributed mutation ratios, where the cardinality of the set is 65,535. We then select a mutation ratio from the set uniformly at random for each fuzzing iteration. The best range was $[0.00001, 0.02]$, which results in 44 bugs from our dataset. This number of bugs was indeed 57.9% of the optimal adaptive case. From the two experiments, we conclude that optimizing the mutation ratio benefits fuzzing in practice.

4.7.3 Distribution of b Values

In this subsection, we answer the following two questions. First, how do we compute b from a crash? Second, what is the distribution of b in crashes of real-world programs?

Recall from §4.4.3, we estimate an optimal mutation ratio r using $\bar{d}$, which depends upon the distribution of b values. To obtain the distribution, we first collected 4,255 distinct pairs of a crashing input and a seed from our previous study [131]. The crashing inputs are gathered by fuzzing a variety of applications that take in a file as an input for over 650 CPU days. The size of the seeds in the dataset ranged from 43B to 31MB, and the average seed size was 954KB. For each crashing input, we computed the Hamming distance from them to the corresponding seeds. The average Hamming distance was 151,721; the median was 11,430; and the standard deviation was 862,055. Notice that the Hamming distance in this case does *not* represent the size of a minimum buggy bitset: it represents the size of a buggy bitset instead.

To compute the size of a minimum buggy bitset (b), we used a delta debugging technique [47, 158] called bug minimization, presented by Householder *et al.* [85]. The idea is simple: given a crashing input and a corresponding seed, bug minimization iteratively restores bits in the crashing input to the original value of the seed, and determines which bit flips are necessary to crash the program. After the minimization, we compute the Hamming distance from each minimized crashing input to its corresponding seed, which is essentially the value of b.

We used the above algorithm in order to compute the distribution of b in the 4,255 distinct crashes that we collected. Figure 4.8 shows the distribution of b values from our dataset. We found that it is enough to flip 9 bits of a seed on average to trigger crashes in our dataset. The median Hamming distance was 6, and the standard deviation was 18. More than 80% of the b values were less than or equal to 10. In addition, we performed the same experiment on our ground truth data. As a result, we obtained the Hamming distance of 5 on average (median 3), and the

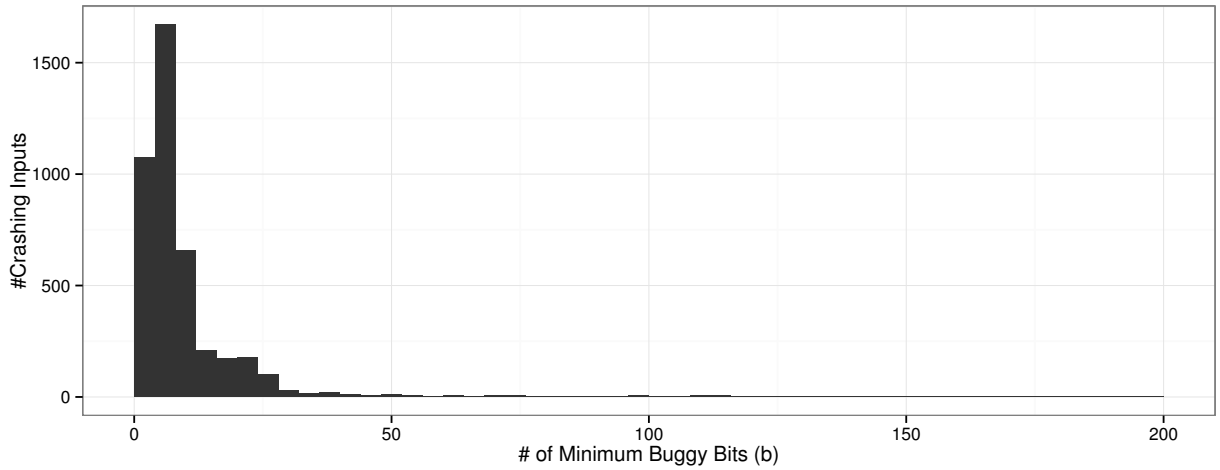


Figure 4.8: The number of minimum buggy bits for 4,255 crashing inputs derived from previous studies. The average was 9 and the median was 6.

standard deviation of the Hamming distance was 10. The result shows that most of the crashes can be triggered by flipping only few bits—less than a byte size in our dataset—from the corresponding seeds.

How Many Bits to Flip? It is important to note that the above result does not necessarily mean that we need to flip only few bits of a seed to effectively trigger program crashes in mutational fuzzing. For example, there may be an input field that is independent from crashes: no matter what value the field has, we can still crash the program. Therefore, in this case, we want to flip more than b bits to increase the likelihood of finding crashes. We indeed found the most number of bugs in `abcm2ps` using a mutation ratio of 0.071, which corresponds to about $2,500$ bit flips. This result highlights the key idea of SymFuzz: a good mutation ratio depends on the input-bit dependence of a seed.

4.7.4 Estimating r

Recall from §4.4.3, the core part of SymFuzz is to derive the average number of dependent bits ($\bar{d}$) from a distribution of b in estimating r. We used Algorithm 4.1 to compute $\bar{d}$ and obtained r for each program. Table 4.5 summarizes the result. The second column of the table shows $\bar{d}$. The third column of the table is the size of the seed N that is used for each of the programs. The fourth column is the number of bugs found using the obtained r for 1 hour of fuzzing. The fifth

Program	$\bar{d}$	Seed Size N	#Bugs	Max. #Bugs	Diff.
abcm2ps	164	35,040	22	24	2
autotrace	69	16,304	10	14	4
bib2xml	484	177,152	2	2	0
catdvi	24	1,632	7	8	1
figtoipe	44	8,016	14	21	7
gif2png	144	1,816	3	3	0
pdf2svg	434	23,368	0	1	1
mupdf	201	23,368	1	3	2

Table 4.5: The number of bugs found with IBDI.

column is the maximum attainable number of bugs in each program for 1 hour of fuzzing when the empirically optimal mutation ratio is selected. The last column is the difference between the number of bugs found with SymFuzz and the optimal number of bugs.

SymFuzz successfully estimated effective mutation ratios for each program, and found 77.6% of the bugs that can be found from the optimal adaptive strategy. Most mutation ratios that we obtained was close to optimal mutation ratios except for the case of `figtoipe`. To investigate the problem, we first ran bug minimization on every unique crash that we obtained for `figtoipe`. We then checked the cardinality of the minimum buggy bitsets (b) for the crashes, and found that $\bar{d}$ was $5\times$ greater than the average input-bit dependence for the minimum buggy bitsets, which results in a smaller mutation ratio than the optimal one. This is a corner case where buggy bits are not close to the other bits in a seed, in which our algorithm can perform poorly.

4.7.5 SymFuzz Practicality

In this subsection, we test the practicality of mutation ratio optimization by comparing the number of bugs found with existing mutational fuzzers such as BFF, `zzuf`, and AFL.

Comparison against BFF and `zzuf`

The closest practical mutational fuzzers in terms of the underlying mutation process are BFF and `zzuf`: they use bit-flipping-based mutation for fuzzing. We fuzzed each of the programs in our dataset for 1 hour using `zzuf`, BFF, and SymFuzz, and compared the number of bugs found. To run `zzuf`, we used a single mutation ratio of 0.004, which is a default ratio. Notice BFF uses dynamic

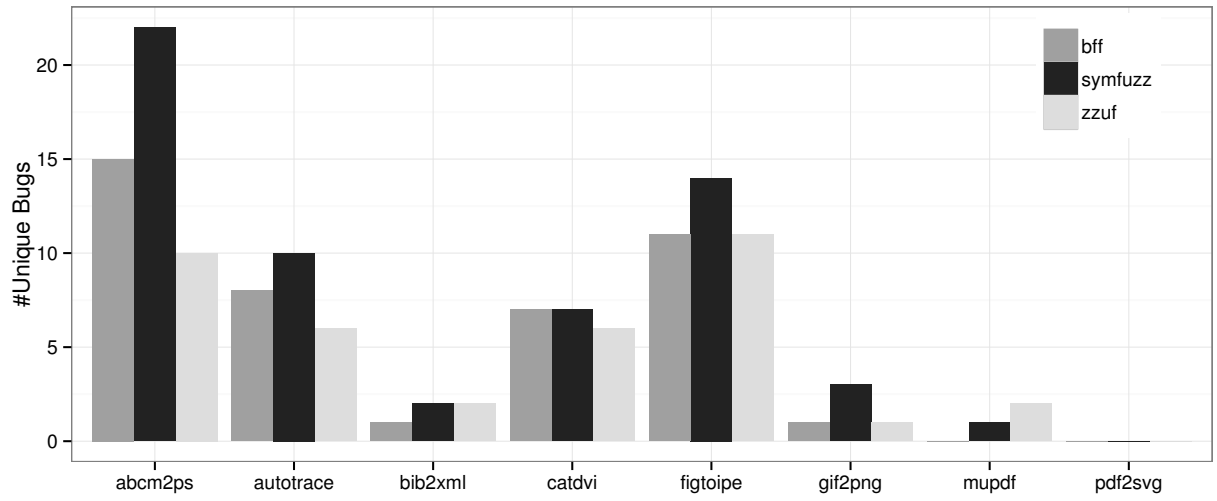


Figure 4.9: Final comparison in the number of bugs found.

scheduling algorithm to automatically find good mutation ratios to use, whereas `zzuf` requires an analyst to specify either a mutation ratio or a range of mutation ratios. In total, BFF found 43 bugs; `zzuf` found 38 bugs; and SymFuzz found 59 bugs. The result indicates that SymFuzz's adaptive strategy found 37.2% and 55.3% more bugs than BFF and `zzuf` respectively. For further analysis, we show a head-to-head comparison against BFF and `zzuf` for each program in Figure 4.9. Notice that SymFuzz found equal or more number of bugs compared to BFF for all the programs. SymFuzz was also superior than `zzuf` except for one program: `mupdf`. The primary reason is because the performance of fuzzing for `mupdf` is sensitive to the mutation ratio from Figure 4.5a. SymFuzz obtained a ratio of 0.003, which is just 0.001 off from the empirically optimal mutation ratio (0.004). `zzuf`'s default mutation ratio happened to be the same as the optimal one.

Comparison against AFL

AFL [156] is the state-of-the-art mutational fuzzer that is used by many security practitioners. The mutation process of AFL consists of two major phases. First, it performs a series of deterministic bit-flipping algorithms based on several heuristics. Second, it uses a random combination of the algorithms in order to non-deterministically generate test cases. These two steps are applied for every seed during a fuzzing campaign. If any one of the generated test cases exhibits a new execution path (based on branch coverage), AFL uses it as a new seed.

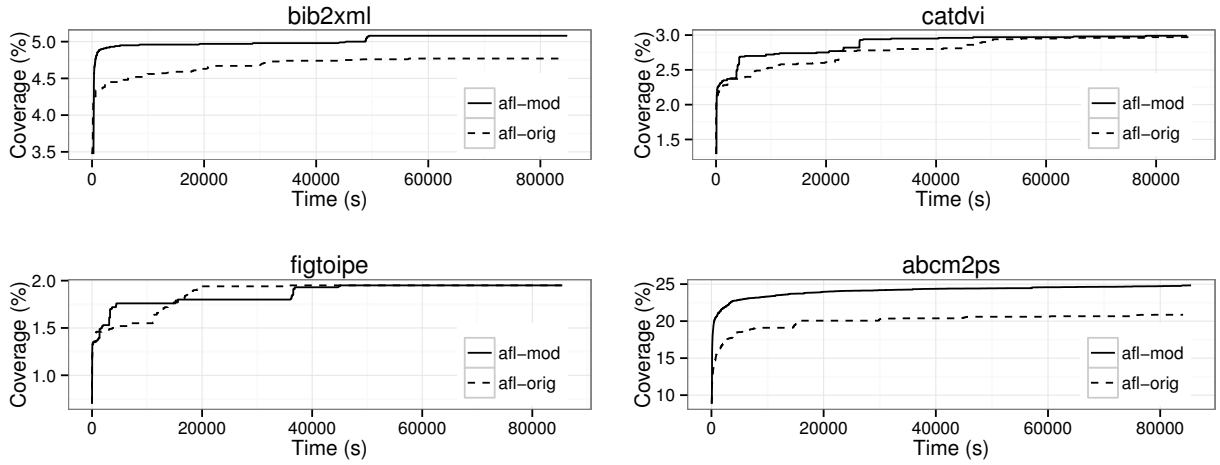


Figure 4.10: Branch coverage difference between AFL and modified AFL (with mutation-ratio-based mutation logic).

Since AFL uses radically different mutation algorithms than SymFuzz, we cannot measure the effectiveness of mutation ratio optimization by directly comparing AFL with SymFuzz. Instead, we replaced the first phase of AFL with SymFuzz's mutation algorithm with mutation ratio optimization, which allows us to compare the effect of using our algorithm over their bit-flipping mutation algorithm. We downloaded AFL 1.45b for this experiment. We ran both the modified AFL and the original AFL on 7 programs (excluding `mupdf` because AFL does not support GUI application) for 24 hours. After 24 hours of fuzzing we triaged all the crashes found using our safe stack hash. As a result, we found 54 bugs from the original AFL, and 64 bugs from the modified AFL. In other words, we found 18.5% more bugs by applying our technique on AFL. We also computed the branch coverage per time during the 24 hours of fuzzing. Figure 4.10 shows the coverage differences in 4 applications that present the most significant differences; we did not observe significant coverage differences from the rest. We conclude that AFL can benefit from our technique in our dataset.

4.8 Discussion

Statistical Significance. Currently, SymFuzz outperforms previous fuzzers in our dataset. However, the result may change with other applications that have different statistical properties in terms of b and d values. Furthermore, our ground truth dataset is based only on fuzzing campaigns of

one hour. Since fuzzing usually runs for several weeks in practice, fuzzing longer would allow a stronger conclusion. We leave fuzzing for more time as future work.

Multiple Mutation Ratios. Two distinct bugs can have significantly different d values, although our current strategy focuses on finding a single $\bar{d}$ from the overall average of d values. This is a fundamental limitation of SymFuzz because we do not know exact minimum buggy bitsets prior to fuzzing. One potential future direction is to consider multiple d values and perform scheduling over the derived mutation ratios.

Seed File. Currently, we assume a seed file for a program is given by an analyst. This is a common assumption for most of the fuzzers in practice. Recent work [131] partially addresses the problem using a coverage-based inference. We leave combining the seed selection algorithm with SymFuzz as future work. Additionally, our analysis only analyzes a single execution path based on a given seed. Therefore, it is possible to miss several input-bit dependence relations that manifest only when a different execution path is taken. Furthermore, our operational semantics (§4.5.1) do not differentiate bit-level operators such as logical-AND from other operators. This may result in an over-approximated results for our analysis, i.e., some bits may have more dependent bits than it is supposed to be. Guaranteeing a bit-level accuracy is out of scope of this work.

Mutational Fuzzing Types. Recall from §2.5, we defined two types of mutational fuzzing methods. Although SymFuzz's primary focus was on surface-based mutational fuzzing, we believe our formulation can also be applied to ball-based mutational fuzzing. As we discussed in §2.5.3, the difference between ball-based and surface-based mutational fuzzing is negligible in practice where the choice of mutation ratios is typically small: the best mutation ratios in our dataset were all less than 0.01.

Bit-Flip-Based Fuzzing. SymFuzz currently focuses on bit-flip-based mutational fuzzing where the size of generated inputs is fixed. Although this constraint simplifies our mathematical model, it is important to support other types of mutational fuzzing for completeness. We leave it as future work to support different types of MBF.

Input-Format-Aware Fuzzing. Although IBDI is inspired by automatic input format recovery [31, 54, 101, 102], our technique currently does not leverage the input field information. One may use the existing input format recovery techniques to find a set of input fields, and mutate a set of specific input fields instead of fuzzing the entire seed file. One potential research direction is to derive the optimal time allocation for each of the input fields in order to maximize the number of bugs found.

4.9 Summary

We designed an algorithm to optimize the mutation ratio in mutational fuzzing given a program and a seed. In particular, we introduced SymFuzz, which runs both black- and white-box analysis to find bugs in a program. We also have formulated the failure rate of mutational fuzzing in terms of the input-bit dependences among bit positions in an input. Our mathematical model led us to design a novel technique for mutation ratio optimization, which estimates a probabilistically optimal mutation ratio from an execution trace. With our data set, we showed that SymFuzz can find 39.5% more bugs than BFF and 57.9% more bugs than `zzuf` in the same amount of fuzzing time. We have also applied our technique to improve AFL. With our modifications, AFL was able to find 18.5% more bugs in the same 24-hour experiment.

Chapter 5

Parameter Scheduling

> Study the past, if you would divine the future.
>
> —Confucius

While the techniques described in Chapter 3 and Chapter 4 help us reducing the number of fuzz configurations to consider, we still need to decide how much time to allocate for each of the reduced configurations. The choice of a fuzz configuration can dramatically affects the number of bugs found from fuzzing. For example, a bug in a program may only be triggered when the input size is greater than 42 bytes. There can be a seed input that allows a mutational fuzzer to find a bug less than a minute, whereas other seed inputs require the same fuzzer to spend more than an hour to find the same bug.

The key question we try to answer in this chapter is how can we organize our time budget to maximize the outcome of fuzzing for a given set of fuzz configurations. Specifically, Which programs should we fuzz? Which fuzz configuration should we use, and in what order? How much time should we dedicate to a fuzz configuration?

5.1 Exploiting Characteristics of Fuzzing Outcome

We exploit the following characteristics of fuzzing outcome in this chapter. First, we view the outcome of mutational fuzzing as a bug arrival process that has diminishing returns. This allows us to model the probability of seeing a new bug in the next trial of fuzz run. Second, mutational fuzzing tends to produce more number of bugs in the future with a configuration that found the

most number of bugs so far. Therefore, one can prioritize a configuration that found the most number of bugs so far in order to maximize the fuzzing outcome given limited time.

5.2 Problem Setting

Recall from §2.1, a fuzz configuration is a set of parameters that a fuzzing algorithm takes in. As a simplification, we assume that a fuzz campaign is orchestrated in *epochs*—an epoch is a sequence of fuzz runs. At the beginning of each epoch, we select one fuzz configuration based only on information obtained during the campaign, and we fuzz that configuration for the entire epoch. This assumption has two subtle but important implications. First, though it does not limit us to fuzzing with only one computer, it does require that every computer in the campaign fuzz the same configuration during an epoch. Second, what we need to select for each epoch is really a fuzz configuration, which gives rise to our naming of the *Fuzz Configuration Scheduling* (FCS) problem.

Fuzz Configuration Scheduling Challenge. Given a list of K fuzz configurations $\{c_1, c_2, \cdots, c_K\}$ and a time budget T, the Fuzz Configuration Scheduling problem seeks to maximize the number of unique bugs discovered in a fuzz campaign that runs for a duration of length T. A *fuzz campaign* is divided into *epochs*, starting with epoch 1. We consider two epoch types: fixed-run and fixed-time. In a fixed-run campaign, each epoch corresponds to a constant number of fuzz runs; since the time required for individual fuzz runs may vary, fixed-run epochs may take variable amounts of time. On the other hand, in a fixed-time campaign, each epoch corresponds to a constant amount of time. Thus, the number of fuzz runs completed may vary across fixed-time epochs.

An online scheduling algorithm Fuzz-Schedule for the Fuzz Configuration Scheduling problem operates before each epoch starts. When the campaign starts, Fuzz-Schedule receives the number K. Suppose the campaign has completed ℓ epochs so far. Before epoch $(\ell + 1)$ begins, Fuzz-Schedule should select a number $i \in [1, K]$ based on the information it has received from the campaign. Then the entire epoch $(\ell + 1)$ is devoted to fuzzing c_i. When the epoch ends, Fuzz-Schedule receives a sequence of IDs representing the outcomes of the fuzz runs completed during the epoch. If an outcome is a crash, then the returned ID is the bug ID computed by the bug triage process, which we assume is non-zero. Otherwise, the outcome is a proper termination, and the

returned ID is 0. Also, any ID that has never been encountered by the campaign prior to epoch $(\ell+1)$ is marked as new. Notice that a new ID can signify either the first proper termination in the campaign or a new bug discovered during epoch $(\ell+1)$. Besides the list of IDs, Fuzz-Schedule also receives statistical information about the epoch. In a fixed-run campaign, it receives the time spent in the epoch; in a fixed-time campaign, it receives the number of fuzz runs that ended inside the epoch.

5.3 Algorithmic Considerations

We now turn to a few technical issues that we withheld from the above problem statement. First, we allow Fuzz-Schedule to be either deterministic or randomized. This admits the use of various existing MAB algorithms, many of which are indeed randomized.

Second, notice that Fuzz-Schedule receives only the number of configurations K but not the actual configurations. This formulation is to prevent Fuzz-Schedule from analyzing the content of any fuzz configurations. Similarly, we prevent Fuzz-Schedule from analyzing bugs by sending it only the bug IDs but not any concrete representation.

Third, Fuzz-Schedule also does not receive the time budget T. This forces Fuzz-Schedule to make its decisions without knowing how much time is left. Therefore, Fuzz-Schedule has to attempt to discover new bugs as early as possible. While this rules out any algorithm that adjusts its degree of exploration based on the time left, we argue that this not a severe restriction from the perspective of algorithm design. For example, one of the algorithms we use is the EXP3.S.1 algorithm [12]. It copes with the unknown time horizon by partitioning time into exponentially longer periods and picking new parameters at the beginning of each period, which has a known length.

Fourth, our analysis assumes that the K fuzz configurations are chosen such that they yield disjoint sets of bugs. This assumption is needed so that we can consider the bug arrival process of fuzzing each configuration independently. While this assumption may be valid when every configuration involves a different program, as in one of our two datasets, satisfying it when one program can appear in multiple configurations is non-trivial. In practice, it is achieved by selecting seeds that exercise different code regions. For example, in our other data set, we use seeds of

various file formats to fuzz the different file parsers in a media player.

Finally, at present we do not account for the time spent in bug triage, though this process requires considerable time. In practice, triaging a crash takes approximately the same amount of time as the fuzz run that initially found the crash. Therefore, bug triage can potentially account for over half of the time spent in an epoch if crashes are extremely frequent. We plan to incorporate this consideration into our project at a future time.

5.4 Multi-Armed Bandits

To find the greatest number of unique bugs given the above problem setting, we must allocate our time wisely. Since initially we have no information on which configuration will yield more new bugs, we should explore the configurations and reduce our risk by fuzzing each configuration for an adequate amount of time. As we start to identify some configurations that we believe may yield more new bugs in the future, we should also exploit this information by increasing the time allocated to fuzz these configurations. Of course, any increase in exploitation reduces exploration, which may cause our analyst to under-explore and miss configurations that are capable of yielding more new bugs. This is the classic "exploration vs. exploitation" trade-off, which signifies that we are dealing with a Multi-Armed Bandit (MAB) problem [22].

This has already been observed by previous researchers. For example, the CERT Basic Fuzzing Framework (BFF) [86], which supports fuzzing a single program with a collection of seeds and a set of mutation ratios, uses an MAB algorithm to select among the seed-ratio pairs during a fuzz campaign. However, we must stress that recognizing the MAB nature of our problem is merely a first step. In particular, we should not expect an MAB algorithm with provably "good" performance, such as one from the UCB [13] or the EXP3 [12] families, to yield good results in our problem setting. There are at least two reasons for this.

First, although many of these algorithms are proven to have optimal *regret* in various forms, the most common form of regret does not actually give good guarantees in our problem setting. In particular, this form of regret measures the difference between the expected reward of an algorithm and the reward obtained by consistently fuzzing the single best configuration that yields the greatest number of unique bugs. However, we are interested in evaluating performance relative to

the total number of unique bugs from *all* K configurations, which may be much greater than the number from one *fixed* configuration. Thus, the low-regret guarantee of many MAB algorithms is in fact measuring against a target that is likely to be much lower than what we desire. In other words, given our problem setting, these algorithms are not guaranteed to be competitive at all.

Second, while there exist algorithms with provably low regret in a form suited to our problem setting, the actual regret bounds of these algorithms often do not give meaningful values in practice. For example, one of the MAB algorithms we use is the EXP3.S.1 algorithm [12], proven to have an expected *worst-case regret* of $\frac{S+2e}{\sqrt{2}-1}\sqrt{2K\ell\ln(K\ell)}$, where S is a certain hardness measure of the problem instance as defined in [12, §8] and ℓ is the number of epochs in our problem setting. Even assuming the easiest case where S equals to 1 and picking K to be a modest value of 10, the value of this bound when $\ell = 4$ is already slightly above 266. However, as we see in §5.9, the number of bugs we found in our two datasets are 200 and 223 respectively. What this means is that this regret bound is very likely to dwarf the number of bugs that can be found in real-world software after a very small number of epochs. In other words, even though we have the right kind of guarantee from EXP3.S.1, the guarantee quickly becomes meaningless in practical terms.

Having said the above, we remark that this simply means such optimal regret guarantees may not be useful in *ensuring* good results. As we will see in §5.9, EXP3.S.1 can still obtain reasonably good results in the right setting.

5.5 Fuzzing as a Weighted CCP

As a first step to design more suitable MAB algorithms for our problem, we discover that the memoryless property of MBF[1] allows us to formally model the repeated fuzzing executions as a bug arrival process. Our insight is that this process is a *weighted* variant of the Coupon Collector's Problem (CCP) where each coupon type has its own fixed but initially *unknown* arrival probability.

As we explained in §2.3, the output of repeated fuzzing executions is a stream of crashes intermixed with proper terminations, which is then transformed into a stream of bug IDs by a triaging process. Since we want to maximize the number of unique bugs found, we are naturally interested in when a new bug arrives in this process. This insight quickly leads us to the Coupon Collector's

[1] Both ball-based and surface-based mutational fuzzing (§2.5) have the memoryless property: any fuzz run is independent of the past fuzz runs.

Problem (CCP), a classical arrival process in probability theory.

The CCP concerns a consumer who obtains one coupon with each purchase of a box of breakfast cereal. Suppose there are M different coupon types in circulation. One basic question about the CCP is: what is the expected number of purchases required before the consumer amasses k ($\leq M$) *unique* coupons? In its most elementary formulation, each coupon is chosen uniformly at random among the M coupon types. In this setting, many questions related to the CCP—including the one above—are relatively easy to answer.

Unfortunately, our problem setting actually demands a weighted variant of the CCP which we dub the WCCP. Intuitively, this is because the probabilities of the different outcomes from a fuzz run are not necessarily (and unlikely to be) uniform. This observation has also been made by Arcuri *et al.* [10].

Let $(M-1)$ be the actual number of unique bugs discoverable by fuzzing a certain configuration. Then including proper termination of a fuzz run as an outcome gives us exactly M distinct outcome types. We thus relate the process of repeatedly fuzzing a configuration to the WCCP by viewing fuzz run outcomes as coupons and their associated IDs as coupon types.

However, unlike usual formulations of the WCCP where the distribution of outcomes across type is given, in our problem setting this distribution is unknown *a priori*. In particular, there is no way to know the true value of M for a configuration without exhaustively fuzzing all possible mutations. As such, we utilize statistical estimations of these distributions rather than the ground-truth in our algorithm design. An important question to consider is whether accurate estimations are feasible.

We now explain why we prefer the sets of bugs from different configurations used in a campaign to be disjoint. Observe that our model of a campaign is a combination of multiple independent WCCP processes. If a bug that is new to one process has already been discovered in another, then this bug cannot contribute to the total number of unique bugs. This means that overlap in the sets of bugs diminishes the fidelity of our model, so that any algorithm relying on its predictions may suffer in performance.

WCCP Notation. Before we go on, let us set up some additional notation related to the WCCP. In an effort to avoid excessive indices, our notation implicitly assumes a fixed configuration c_i that

is made apparent by context. For example, M, the number of possible outcomes when fuzzing a given configuration as defined above, follows this convention.

(i) Consider the fixed sequence σ of outcomes we obtain in the course of fuzzing c_i during a campaign. We label an outcome as type k if it belongs to the k^{th} distinct type of outcome in σ. Let $\mathcal{P}_k$ denote the probability of encountering a type-k outcome in σ, i.e.,

$$\mathcal{P}_k = \frac{\left|\{x \in \mathcal{N}_{\lfloor |s_i| \cdot r \rfloor}(s_i) : x \text{ triggers an outcome of type } k\}\right|}{\left|\mathcal{N}_{\lfloor |s_i| \cdot r \rfloor}(s_i)\right|}. \tag{5.1}$$

(ii) Although both the number and frequency of outcome types obtainable by fuzzing c_i are unknown a priori, during a campaign we do have empirical observations for these quantities up to any point in σ. Let $\hat{M}(\ell)$ be the number of distinct outcomes observed from epoch 1 through epoch ℓ. Let $n_k(\ell)$ be the number of inputs triggering outcomes of type k observed throughout these ℓ epochs. Notice that over the course of a campaign, the sequence σ is segmented into subsequences, each of which corresponds to an epoch in which c_i is chosen. Thus, the values of $\hat{M}(\cdot)$ and $n_k(\cdot)$ will not change if c_i is not chosen for the current epoch. With this notation, we can also express the empirical probability of detecting a type-k outcome following epoch ℓ as

$$\hat{\mathcal{P}}_k(\ell) = \frac{n_k(\ell)}{\sum_{k'=1}^{\hat{M}(\ell)} n_{k'}(\ell)}.$$

5.6 Impossibility Results

The absence of any assumption on the distribution of outcome types in the WCCP quickly leads us to our first impossibility result. In particular, *no* algorithm can consistently outperform other algorithms for the FCS problem. This follows from a well-known impossibility result in optimization theory, namely the "No Free Lunch" theorem by Wolpert and Macready [149]. Their theorem implies that "any two optimization algorithms are equivalent when their performance is averaged across all possible problems." In our problem setting, maximizing the number of bugs found in epoch $(\ell+1)$ amounts to, for each configuration, estimating its $\mathcal{P}_{\hat{M}(\ell)+1}$ in equation (5.1) using only past observations from that configuration. Intuitively, by averaging across all possible outcome type distributions, any estimation will be incorrect *sufficiently often* and thus lead to suboptimal

behavior that cancels any advantage of one algorithm over another.

While we may consider this result to be easy to obtain once we have properly set up our problem, we consider it to be an important intellectual contribution for the pragmatic practitioners who remain confident that they *can* design algorithms that outperform others. In particular, the statement of the No Free Lunch theorem itself reveals precisely how we can circumvent its conclusion—our estimation procedure must *assume* the outcome type distributions have particular characteristics. Our motto is thus "there is no free lunch—please bring your own prior!"

Our second impossibility result shows that there are problem instances in which the time spent by any deterministic online algorithm to find a given number of unique bugs in a fixed-time campaign is at least K times larger than the time spent by an optimal offline algorithm. Using the terminology of competitive analysis, this shows that the competitive ratio of any deterministic online algorithm for this problem is at least K.

To show this, we fix a deterministic algorithm A and construct a contrived problem instance in which there is only *one* bug among all the configurations in a campaign. Since A is deterministic, there exists a unique configuration c_i^* that gets chosen *last*. In other words, the other $(K-1)$ configurations have all been fuzzed for at least one epoch when c_i^* is fuzzed for the first time. If the lone bug is only triggered by fuzzing c_i^*, then A will have to fuzz for at least K epochs to find it. For an optimal offline algorithm, handling this contrived scenario is trivial. Since it is offline, it has full knowledge of the outcome distributions, enabling it to hone in on the special configuration c_i^*) and find the bug in the first epoch. This establishes that K is a lowerbound for the competitive ratio of any deterministic algorithm. Finally, we observe that Round-Robin is a deterministic online algorithm that achieves the competitive ratio K in *every* problem instance. It follows immediately that K is tight.

5.7 Scheduling Algorithm Design

Having seen such strong impossibility results, let us consider what a pragmatist might do before bringing in any prior on the outcome type distribution. In other words, if we do not want to make any assumptions on this distribution, is there a justifiable approach to designing online algorithms for the FCS problem?

We argue that the answer is yes. Consider two fuzz configurations c_1 and c_2 for which we have *upperbounds* on the probability of finding a new outcome if we fuzz them once more. Assume that the upperbound for c_1 is the higher of the two. We stress that what we know are merely upperbounds—it is still possible that the true probability of yielding a new outcome from fuzzing c_1 is lower than that of c_2. Nonetheless, with no information beyond the ordering of these upperbounds, fuzzing c_1 first is arguably the more prudent choice. This is because to do otherwise would indicate a belief that the actual probability of finding a new outcome by fuzzing c_1 in the next fuzz run is lower than the upperbound for c_2. Accepting this argument, how might we obtain such upperbounds? We introduce the Rule of Three for this purpose.

5.7.1 Rule of Three

Consider an experiment of independent Bernoulli trials with identical success and failure probabilities p and $q = (1 - p)$. Suppose we have carried out $N \geq 1$ trials so far and every trial has been a success. What can we say about q other than the fact that it must be (i) at least 0 to be a valid probability and (ii) strictly less than 1 since p is evidently positive? In particular, can we place a lower upperbound on q?

Unfortunately, the answer is a resounding *no*: even with q arbitrarily close to 1, we still have $(p^N > 0)$. This means our observation really *could* have happened even if it is extremely unlikely.

Fortunately, if we are willing to rule out the possibility of encountering extremely unlikely events, then we may compute a lower upperbound for q by means of a confidence interval. For example, a 95% confidence interval on q outputs an interval that includes the true value of q of the underlying experiment with 95% certainty. In other words, if the outputted interval does not contain the true value of q for the experiment, then the observed event must have a likelihood of at most 5%.

For the above situation, there is particularly neat technique to compute a 95% confidence interval on q. Known as the "Rule of Three", this method simply outputs 0 and $3/N$ for the lowerbound and upperbound, respectively. The lowerbound is trivial, and the upperbound has been shown to be a good approximation for $N > 30$. See [91] for more information on this technique, including the relationship between 95% confidence and the constant 3.

How We Use Rule of Three. In order to apply the Rule of Three, we must adapt our fuzzing experiments with any $M > 1$ possible outcome types to fit the mold of Bernoulli trials. We make use of a small trick. Suppose we have just finished epoch ℓ and consider a particular configuration c_1. Using our notation, we have observed $\hat{M}(\ell)$ different outcomes so far and for $1 \leq k \leq \hat{M}(\ell)$, we have observed $n_k(\ell)$ counts of outcome of type k. Let $N(\ell) = \sum_{k=1}^{\hat{M}(\ell)} n_k(\ell)$ denote the total number of fuzz runs for this configuration through epoch ℓ. The trick is to define a "success" to be finding an outcome of type 1 through type $\hat{M}(\ell)$. Then, in hindsight, it *is* the case that our experiment has only yielded success so far.

With this observation, we may now apply the Rule of Three to conclude that $[0, 3/N(\ell)]$ is a 95% confidence interval on the "failure" probability—the probability that fuzzing this configuration will result in an outcome type that we have *not* seen before, i.e., a new outcome. Then, as desired, we have an easy-to-compute upperbound on the probability of finding a new outcome for each configuration. We introduce one more piece of notation before proceeding: define the Remaining Probability Mass (RPM) of c_i at the end of epoch ℓ, denoted $\mathrm{RPM}(\ell)$, to be the probability of finding a new outcome if we fuzz c_i once more. Note that the pair in $\mathrm{RPM}(\ell)$ is implicit, and that this value is upperbounded by $3/N(\ell)$ if we accept a 95% confidence interval.

5.7.2 Design Space

In this section, we explore the design space that a pragmatist may attempt when designing online algorithms for the FCS problem. Our focus here is to explain our motivation for choosing the three dimensions we explore and the particular choices we include in each dimension. By combining these dimensions, we obtain 26 online algorithms for our problem. We implemented these algorithms inside our simulator, FuzzSim, the detail of which is presented in §5.8.

Epoch Type

We consider two possible definitions of an epoch in a fuzz campaign. The first is the more traditional choice and is used in the current version of CERT BFF v2.6 [86]; the second is our proposal.

- **Fixed-Run.** Each epoch executes a constant number of fuzz runs. In FuzzSim, a fixed-run epoch consists of 200 runs. Note that any differential in fuzzing speed across configurations

translates into variation in the time spent in fixed-run epochs.

- **Fixed-Time.** Each epoch is allocated a fixed amount of time. In FuzzSim, a fixed-time epoch lasts for 10 seconds. Our motivation to investigate this epoch type is to see how heavily epoch time variation affects the results obtained by systems with fixed-run epochs.

Belief Metrics

Two of the MAB algorithms we present below make use of a belief metric that is associated with each configuration and is updated after each epoch. Intuitively, the metrics are designed such that fuzzing a configuration with a higher metric *should* yield more bugs in expectation. The first two beliefs below use the concept of RPM to achieve this without invoking any prior; the remaining three embrace a "bug prior". For now, suppose epoch ℓ has just finished and we are in the process of updating the belief for the configuration c_i.

- **RPM.** We use the upperbound in the 95% confidence interval given by the Rule of Three to approximate $\mathrm{RPM}(\ell)$. The belief is simply $3/N(\ell)$.

- **Expected Waiting Time Until Next New Outcome (EWT).** Since RPM does not take into account of the speed of each fuzz run, we also investigate a speed-normalized variant of RPM. Let $\textsc{Time}(\ell)$ be the cumulative time spent fuzzing this configuration from epoch 1 to epoch ℓ. Let $\textsc{avgTime}(\ell)$ be the average time of a fuzz run, i.e., $\frac{\textsc{Time}(\ell)}{N(\ell)}$. Let W be a random variable denoting the waiting time until the next new outcome. Recall that $\mathrm{RPM}(\ell)$ is the probability of finding a new outcome in the next fuzz run and assume it is independent of $\textsc{avgTime}(\ell)$. To compute $E[W]$, observe that either we find a new outcome in the next fuzz run, or we do not and we have to wait again. Therefore,

$$\begin{aligned} E[W] = \mathrm{RPM}(\ell) \times \textsc{avgTime}(\ell) \\ + (1 - \mathrm{RPM}(\ell)) \times (\textsc{avgTime}(\ell) + E[W]). \end{aligned}$$

  (Notice that RPM does not change even in the second case; what changes is our upperbound on RPM.) Solving for $E[W]$ yields $\frac{\textsc{avgTime}(\ell)}{\mathrm{RPM}(\ell)}$, and we substitute in the upperbound of the 95% confidence interval for $\mathrm{RPM}(\ell)$ to obtain $E[W] \geq \frac{\textsc{avgTime}(\ell)}{3/N(\ell)} = \frac{\textsc{Time}(\ell)}{3}$. Since a larger waiting

time is less desirable, the belief used is its reciprocal, $3/\textsc{Time}(\ell)$.

- **Rich Gets Richer (RGR).** This metric is grounded in what we call the "bug prior", which captures our empirical observation that code tends to be either robust or bug-ridden. Programs written by programmers of different skill levels or past testing of a program might explain this real-world phenomenon. Accordingly, demonstrated bugginess of a program serves as a strong indicator that more bugs will be found in that program and thus the belief is $\hat{M}(\ell)$.

- **Density.** This is a runs-normalized variant of RGR and is also the belief used in CERT BFF v2.6 [86]. The belief function is $\hat{M}(\ell)/N(\ell)$. Observe that this is the belief function of RPM scaled by $\hat{M}(\ell)/3$. In other words, Density can be seen as RPM adapted with the bug prior.

- **Rate.** This is a time-normalized variant of RGR. The belief function is $\hat{M}(\ell)/\textsc{Time}(\ell)$. Similar to Density, Rate can be seen as EWT adapted with the bug prior.

Bandit Algorithms

Since the FCS problem is an instance of an MAB problem, naturally we explore a number of MAB algorithms.

- **Round-Robin.** This simply loops through the configurations in a fixed order, dedicating one epoch to each configuration. Note that Round-Robin is a non-adaptive, deterministic algorithm.

- **Uniform-Random.** This algorithm selects uniformly at random from the set of configurations for each epoch. Like Round-Robin, this algorithm is non-adaptive; however, it is randomized.

- **Weighted-Random.** Configurations are selected at random in this algorithm, with the probability associated with each configuration is linked to the belief metric in use. The weight of a well-performing configuration is adjusted upward via the belief metric, thereby increasingly the likelihood of selecting that configuration in future epochs. This mechanism functions in reverse for configurations yielding few or no bugs.

- **ϵ-Greedy.** The ϵ-Greedy algorithm takes an intuitive approach to the exploration vs. exploitation trade-off inherent to MAB problems. With probability ϵ, the algorithm selects a

configuration uniformly at random for exploration. With probability $(1 - \epsilon)$, it chooses the configuration with the highest current belief, allowing it to exploit its current knowledge for gains. The constant ϵ serves as a parameter balancing the two competing goals, with higher ϵ values corresponding to a greater emphasis on exploration.

- **EXP3.S.1.** This is an advanced MAB algorithm by Auer *et al.* [12] for the non-stochastic MAB problem. We picked this algorithm for three reasons. First, it is from the venerable EXP3 family, and so likely to be picked up by practitioners. Second, this is one of the EXP3 algorithms that is not parameterized by any constants and thus no parameter tuning is needed. Third, this algorithm is designed to have an optimal worst-case regret, which is a form of regret that suits our problem setting. Note that at its core EXP3.S.1 is a weighted-random algorithm. However, since we do not have a belief metric that corresponds to the one used in EXP3.S.1, we did not put it inside the Weighted-Random group.

Out of budgetary constraints, we have taken a simulation approach so that we can replay the events from previous fuzzings to try out new algorithms. Since we have recorded all the events that may happen during any fuzz campaign of the same input configurations, we can even attempt to compute what an optimal offline algorithm would do and compare the results of our algorithms against it. In the case when the configurations do not yield duplicated bugs, such as in our Inter-Program dataset (§5.9), we devise a pseudo-polynomial time algorithm that computes the offline optimal. In the other case where duplicated bugs are possible, we propose a heuristic to post-process the solution from the above algorithm to obtain a lowerbound on the offline optimal.

Assuming that the sets of unique bugs from different configurations are disjoint, our algorithm is a small variation on the dynamic programming solution to the Bounded Knapsack problem. Let K be the number of configurations and B be the total number of unique bugs from all K configurations. Let $t(i, b)$ be the minimum amount of time it takes for configuration i to produce b unique bugs. Note that $t(i, b)$ is assumed to be ∞ when configuration i never produces b unique bugs in our dataset. We claim that $t(i, b)$ can be pre-computed for all $i \in [1, K]$ and $b \in [0, B]$, where each entry takes amortized $O(1)$ time given how events are recorded in our system.

Let $m(i, b)$ be the minimum amount of time it takes for configurations 1 through i to produce b unique bugs. We want to compute $m(K, b)$ for $b \in [0, B]$. By definition, $m(1, b) = t(1, b)$ for

$b \in [0, B]$. For $i > 1$, observe that $m(i, b) = \min_{c \in [0,B]}\{t(i, c) + m(i - 1, b - c)\}$. This models partitioning the b unique bugs into c unique bugs from configuration i and $(b - c)$ unique bugs from configurations 1 through $(i - 1)$. Computing each $m(i, b)$ entry takes $O(B)$ time. Since there are $O(K \times B)$ entries, the total running time is $O(K \times B^2)$.

The above algorithm is incorrect when the sets of unique bugs from different configurations are not disjoint. This is because the recurrence formula of $m(i, b)$ assumes that the c unique bugs from configuration i are different from the $(b - c)$ unique bugs from configurations 1 through $(i - 1)$. In this case, we propose a heuristic to compute a lowerbound on the offline optimal. After obtaining the $m(i, b)$ table from the above, we post-process bug counts by the following discount heuristic. First, we compute the maximum number of bugs that can be found at each time by the above algorithm by examining the K-th row of the table. Then, by scanning forward from time 0, whenever the bug count goes up by one due to a duplicated bug (which must have been found using another configuration), we discount the increment. Since the optimal offline algorithm can also pick up exactly the same bugs in the same order as the dynamic programming algorithm, our heuristic is a valid lowerbound on the maximum number of bugs that an optimal offline algorithm would find.

We formalize the notion of ex post facto optimality seed selection and give the strategy that provides an optimal algorithm even if the bugs found by different configurations are correlated in Chapter 3.

5.8 Design & Implementation of FuzzSim

This section presents FuzzSim, a replay-based simulation system for mutational fuzzing that is designed to run different configuration scheduling algorithms using logs from previous fuzzings. FuzzSim employs a three-step approach: (1) fuzzing, (2) triage, and (3) simulation.

Fuzzing. The first step is fuzzing and collecting run logs from a fuzzer. FuzzSim takes in a list of program-seed pairs (p_i, s_i) and a time budget T. It runs a fuzzer on each configuration for the full length of the time budget T and writes to the log each time a crash occurs. Log entries are recorded as 5-tuples of the form (p_i, s_i, time stamp, #runs, mutation identifier).

In our implementation, we fuzz with `zzuf`, one of the most popular open-source fuzzers. `zzuf` uses an approximated version of surface-based mutational fuzzing algorithm. The randomization in `zzuf` can be reproduced given the mutation identifier, thus enabling us to reproduce a crashing input from its seed file and the log entry associated with the crash. For example, an output tuple of (`FFMpeg`, a.avi, 100, 42, 1234) specifies that the program `FFMpeg` crashed at the 100-th second with an input file obtained from "a.avi" according to the mutation identifier 1234. Interested readers may refer to `zzuf` [98] for details on mutation identifiers and the actual implementation.

The deterministic nature of `zzuf` allows FuzzSim to triage bugs after completing all fuzz runs first. In other words, FuzzSim does not compute bug identifiers during fuzzing and instead re-derives them using the log. This does not affect any of our algorithms since none of them relies on the actual IDs. In our experiments, we have turned off address space layout randomization (ASLR) in both the fuzzing and the triage steps in order to reproduce the same crashes.

Triage. The second step of FuzzSim maps crashing inputs found during fuzzings into bugs. At a high level, the triage phase takes in the list of 5-tuples (p_i, s_i, time-stamp, #runs, mutation identifier) logged during the fuzzing step and outputs a new list of 5-tuples of the form (p_i, s_i, time-stamp, #runs, bug identifier). More specifically, FuzzSim replays each recorded crash under a debugger to collect stack traces. If FuzzSim does not detect a crash during a particular replay, then we classify that test case to be a non-deterministic bug and discard it.

We then use the collected stack traces to produce bug identifiers, essentially hashes of the stack traces. In particular, we use the fuzzy stack hash algorithm [122], which identifies bugs by hashing the normalized line numbers from a stack trace. With this algorithm, the number of stack frames to hash has a significant influence on the accuracy of bug triage. For example, taking the full stack trace often leads to mis-classifying a single bug into multiple bugs, whereas taking only the top frame can easily lead to two different bugs being mis-classified as one. To match the state of the art, FuzzSim uses the top 3 frames as suggested in [122]. We propose a variant of this technique that reduces false positives in §4.6.3.

Simulation. The last step simulates a fuzz campaign on the collected ground-truth data from the previous steps using a user-specified scheduling algorithm. More formally, the simulation

Algorithm 5.1: FuzzSim algorithms.

`Fuzzing`: $(\{(p_i, s_i)\}, T)$
→ $\{p_i, s_i$, timestamp, #runs, mutation id$\}$

`Triage`: $\{(p_i, s_i$, timestamp, #runs, mutation id$)\}$
→ $\{(p_i, s_i$, timestamp, #runs, bug id$)\}$

`Simulation`: $\{(p_i, s_i$, timestamp, #runs, bug id$)\}$
→ {(timestamp, #bugs)}

step takes in a scheduling algorithm and a list of 5-tuples of the form (p_i, s_i, timestamp, #runs, bug identifier) and outputs a list of 2-tuples (timestamp, #bugs) that represent the accumulated time before the corresponding number of unique bugs are observed under the given scheduling algorithm.

Since FuzzSim can simulate any scheduling algorithm in an offline fashion using the pre-recorded ground-truth data, it enables us to efficiently compare numerous scheduling algorithms without actually running a large number of fuzz campaigns. During replay, FuzzSim outputs a timestamp whenever it finds a new bug. Therefore, we can easily plot and compare different scheduling algorithms by comparing the number of bugs produced under the same time budget. We summarize FuzzSim's three-step algorithm in Algorithm 5.1.

5.9 FuzzSim Evaluation

To evaluate the performance of the 26 algorithms presented in §5.7.2, we focus on the following questions:

1. Which scheduling algorithm works best for our datasets?
2. Why does one algorithm outperform the others?
3. Which of the two epoch types—fixed-run or fixed-time—works better, and why?

5.9.1 Experimental Setup

Our experiments were performed on Amazon EC2 instances that have been configured with a single Intel 2GHz Xeon CPU core and 4GB RAM each. We used the most recent Debian Linux

Dataset	#runs	#crashes	#bugs
Intra-program	636,998,978	906,577	200
Inter-program	4,868,416,447	415,699	223

Table 5.1: Statistics from fuzzing the two datasets.

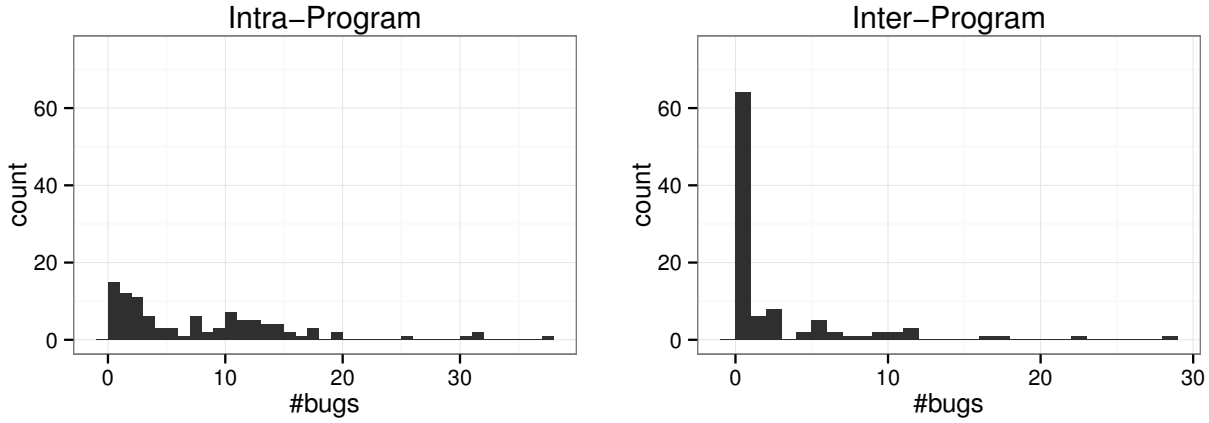


Figure 5.1: Distribution of the number of bugs per configuration in each dataset.

distribution at the time of our experiment (April 2013) and downloaded all programs from the then-latest Debian Squeeze repository. Specifically, the version of FFMpeg we used is SVN-r0.5.10-4:0.5.10-1, which is based on a June 2012 FFMpeg release with Debian-specific patches.

5.9.2 Fuzzing Data Collection

Our evaluation makes use of two datasets: (1) FFMpeg with 100 different input seeds, and (2) 100 different Linux applications, each with a corresponding input seed. We refer to these as the "intra-program" and the "inter-program" datasets respectively.

For the intra-program dataset, we downloaded $10,000$ video/image sample files from the MPlayer website at `http://samples.mplayerhq.hu/`. From these samples, we selected 100 files uniformly at random and took them as our input seeds. The collected seeds include various audio and video formats such as ASF, QuickTime, MPEG, FLAC, etc. We then used `zzuf` to fuzz FFMpeg with each seed for 10 days.

For the inter-program dataset, we downloaded 100 different file conversion utilities in Debian. To select these 100 programs, we first enumerated all file conversion packages tagged as "use::converting" in the Debian package tags interface (debtags). From this list of packages, we

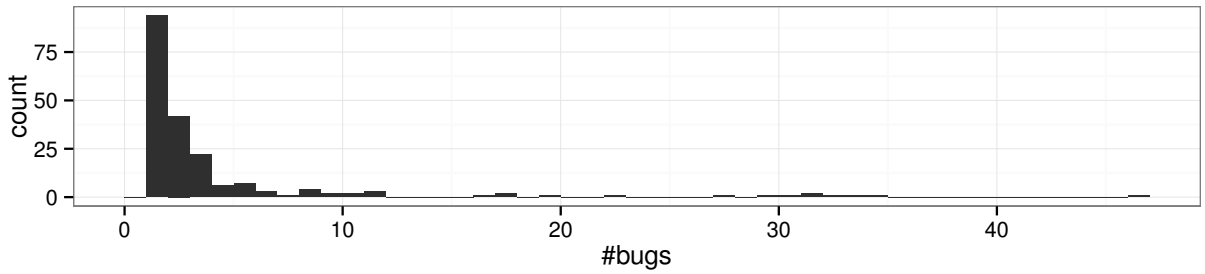


Figure 5.2: Distribution of bug overlaps across multiple seeds for the intra-program dataset.

manually identified 100 applications that take a file name as a command line argument. Then we manually constructed a valid seed for each program and the actual command line to run it with the seed. After choosing these 100 program-seed pairs, we fuzzed each for 10 days as well. In total, we have spent 48,000 CPU hours fuzzing these 200 configurations.

To perform bug triage, we identified and re-ran every crashing input from the log under a debugger to obtain stack traces for hashing. After triaging with the fuzzy stack hash algorithm, we found 200 bugs from the intra-program dataset and 223 bugs from the inter-program dataset. Table 5.1 summarizes the data collected from our experiments. The average fuzzing throughput was 8 runs per second for the intra-program dataset and 63 runs per second for the inter-program dataset. This difference is due to the higher complexity of FFMpeg when compared to the programs in the inter-program dataset.

5.9.3 Data Analysis

What does the collected fuzzing data look like? We studied our data from fuzzing and triage to answer two questions: (1) How many bugs does a configuration trigger? (2) How many bugs are triggered by multiple seeds in the intra-program dataset?

We first analyzed the distribution of the number of bugs in the two datasets. On average, the intra- and the inter-program datasets yielded 8.2 and 2.4 bugs per configuration respectively. Figure 5.1 shows two histograms, each depicting the number of occurrences of bug counts. There is a marked difference in the distributions from the two datasets: 64% of configurations in the inter-program dataset produce no bugs, whereas the corresponding number in the intra-program dataset is 15%. We study the bias of the bug count distribution in §5.9.4.

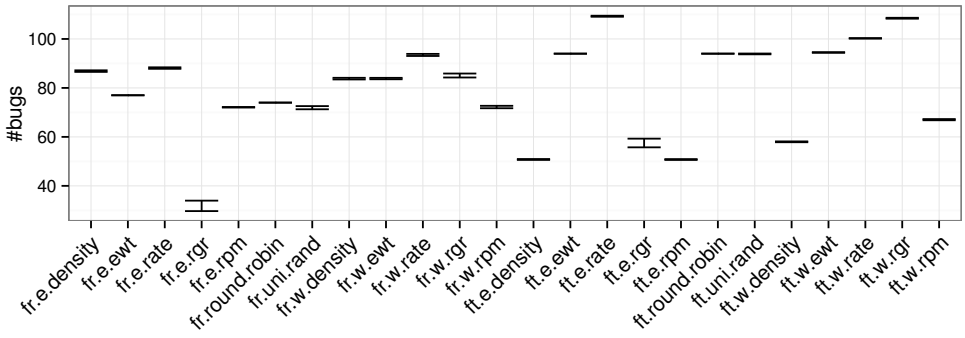
Dataset	Epoch	MAB algorithm	#bugs found for each belief				
			RPM	EWT	Density	Rate	RGR
Intra-Program	Fixed-Run	ϵ-Greedy	72	77	87	88	32
		Weighted-Random	72	84	84	**93**	85
		Uniform-Random			72		
		EXP3.S.1			58		
		Round-Robin			74		
	Fixed-Time	ϵ-Greedy	51	94	51	**109**	58
		Weighted-Random	67	94	58	100	108
		Uniform-Random			94		
		EXP3.S.1			95		
		Round-Robin			94		
Inter-Program	Fixed-Run	ϵ-Greedy	90	119	89	89	41
		Weighted-Random	90	131	92	**135**	94
		Uniform-Random			89		
		EXP3.S.1			72		
		Round-Robin			90		
	Fixed-Time	ϵ-Greedy	126	158	111	164	117
		Weighted-Random	152	157	100	**167**	165
		Uniform-Random			158		
		EXP3.S.1			161		
		Round-Robin			158		

Table 5.2: Comparison between scheduling algorithms.

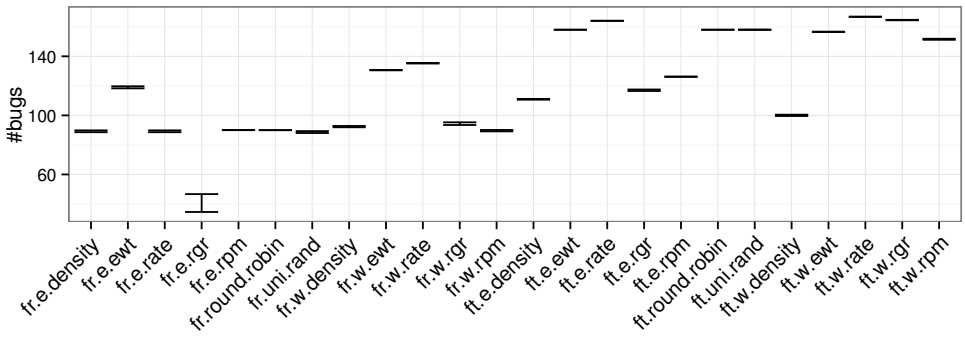
Second, we measured how many bugs are shared across seeds in the intra-program dataset. As an extreme case, we found a bug that was triggered by 46 seeds. The average number of seeds leading to a given bug is 4. Out of the 200 bugs, 97 were discovered from multiple seeds. Figure 5.2 illustrates the distribution of bug overlaps. Our results suggest that there is a small overlap in the code exercised by different seed files even though they have been chosen to be of different types. Although this shows that our bug disjointness assumption in the WCCP model does not always hold in practice, the low average number of seeds leading to a given bug in our dataset means that the performance of our algorithms should not have been severely affected.

5.9.4 Simulation

We now compare the 26 scheduling algorithms based on the 10-day fuzzing logs collected for the intra- and inter-program datasets. To compare the performance of scheduling algorithms, we use the total number of unique bugs reported by the bug triage process. Recall from §5.7.2 that these algorithms vary across three dimensions: (1) epoch types, (2) belief metrics, and (3) MAB algo-



(a) Intra-program.



(b) Inter-program.

Figure 5.3: The average number of bugs over 100 runs for each scheduling algorithm with error bars showing a 99% confidence interval. "ft" represents fixed-time epoch; "fr" represents fixed-run epoch; "e" represents ϵ-Greedy; "w" represents Weighted-Random.

rithms. For each valid combination (see Table 5.2), we ran our simulator 100 times and averaged the results to study the effect of randomness on each scheduling algorithm. In our experiments, we allocated 10 seconds to each epoch for fixed-time campaigns and 200 runs for fixed-run campaigns. For the ϵ-Greedy algorithm, we chose ϵ to be 0.1.

Table 5.2 summarizes our results. Each entry in the table represents the average number of bugs found by 100 simulations of a 10-day campaign. We present ϵ-Greedy and Weighted-Random at the top of each epoch-type row group, each showing five entries that correspond to the belief metric used. For the other three MAB algorithms, we only show a single entry in the center because these algorithms do not use our belief metrics. Figure 5.3 describes the variability of our data using error bars showing a 99% confidence interval. Notice that 94% of our scheduling algorithms have a confidence interval that is less than 2 (bugs). RGR gives the most volatile algorithms. This is not surprising because RGR tends to under-explore by focusing too much on bug-yielding config-

urations that it encounters early on in a campaign. In the remainder of this section, we highlight several important aspects of our results.

Fixed-time algorithms prevail over fixed-run algorithms. In the majority of Table 5.2, except for RPM and Density in the intra-program dataset, fixed-time algorithms always produced more bugs than their fixed-run counterparts. Intuitively, different inputs to a program may take different amounts of time to execute, leading to different fuzzing throughputs. A fixed-time algorithm can exploit this fact and pick configurations that give higher throughputs, ultimately testing a larger fraction of the input space and potentially finding more bugs. To investigate the above exceptions, we have also performed further analysis on the intra-program dataset. We found that the performance of the fixed-time variants of RPM and Density greatly improves in longer simulations. In particular, *all* fixed-time algorithms outperform their fixed-run counterparts after day 11.

Along the same line, we observe that fixed-time algorithms yield $1.6\times$ more bugs on average when compared to their fixed-run counterparts in the inter-program dataset. In contrast, the improvement is only $1.1\times$ in the intra-program dataset. As we have explained above, fixed-time algorithms tend to perform more fuzz runs and potentially finding more bugs by taking advantage of faster configurations. Thus, if the runtime distribution of fuzz runs is more biased, as in the case of the inter-program dataset, then fixed-time algorithms tend to gain over their fixed-run counterparts.

Time-normalization outperforms runs-normalization. In our results, EWT always outperforms RPM and Rate always outperforms Density. We believe that this is because EWT and Density do not spend more time on slower programs and slower programs are not necessarily buggier. The latter hypothesis seems highly plausible to us; if true, it would imply that time-normalized belief metrics are more desirable than runs-normalized metrics.

Fixed-time Rate works best. In both datasets, the best-performing algorithms use fixed-time epochs and Rate as belief (entries shown in boldface in Table 5.2). Since Rate can be seen as a time-normalized variant of RGR, this gives further evidence of the superiority of time normalization. In addition, it also supports the plausibility of the bug prior.

5.9.5 Speed of Bug Finding

Besides the number of bugs found at the end of a fuzz campaign, the speed at which bugs are discovered is also an important metric for evaluating scheduling algorithms. We address two questions in this section. First, is there a scheduling algorithm that prevails throughout an entire fuzz campaign? Second, how effective are the algorithms with respect to our offline algorithm in §5.7.2? To answer the questions, we first show the speed of each algorithm in Figure 5.4 and Figure 5.5 by computing the number of bugs found over time. For brevity and readability, we picked for each belief metric the algorithm that produced the greatest average number of unique bugs at the end of the 10-day simulations.

Speed. We observe that Rate and RGR are in the lead for the majority of the time during our 10-day simulations. In other words, not only do they find more unique bugs at the end of the simulations, but they also outperform other algorithms at almost any given time. This lends further credibility to the bug prior.

Effectiveness. We also compare the effectiveness of each algorithm by observing how it compares against our offline algorithm. We have implemented the offline algorithm including the post-processing step that discounts duplicated bugs and computed the solution for each dataset. The numbers of bugs found by the offline algorithm for the intra- and the inter-program datasets are 132 and 217 respectively. (Notice that due to bug overlaps and the discount heuristic, these are lowerbounds on the offline optimal.) As a comparison, Rate found 83% and 77% of bugs in the intra- and inter-program datasets, respectively. Based on these numbers, we conclude that Rate-based algorithms are effective.

5.9.6 Comparison with CERT BFF

At present, the CERT Basic Fuzzing Framework (BFF) [86] is the closest system that makes use of scheduling algorithms for fuzz campaigns. In this section, we evaluate the effectiveness of BFF's scheduling algorithm using our simulator.

Based on our study of the source code of BFF v2.6 (the latest version as of this writing), it uses a fixed-run weighted-random algorithm with Density ($\frac{\#bugs}{\#runs}$) as its belief metric. However,

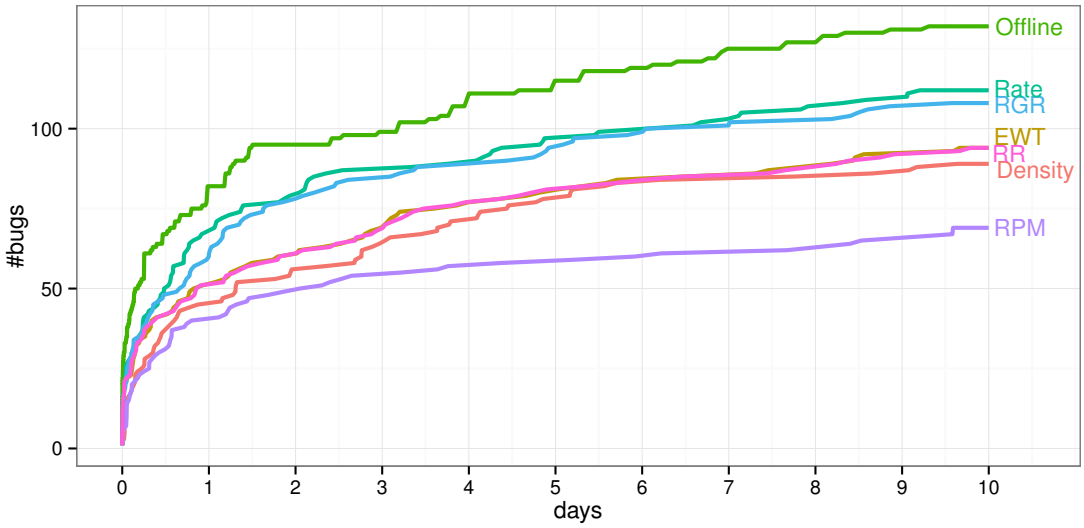


Figure 5.4: Bug finding speed of different belief-based algorithms for the intra-program dataset.

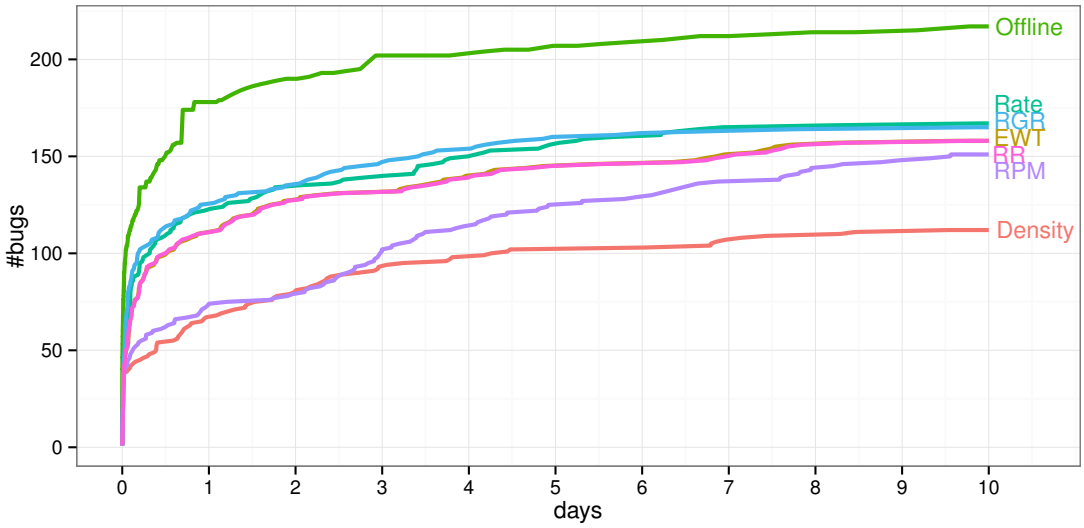


Figure 5.5: Bug finding speed of different belief-based algorithms for the inter-program dataset.

a key feature of BFF prevented us from completely implementing its algorithm in our simulation framework. In particular, while BFF focuses on fuzzing a single program, it considers not only a collection of seeds but also a set of predetermined mutation ratios. In other words, instead of choosing program-seed pairs as in our experiments, BFF chooses seed-ratio pairs with respect to a single program. Since our simulator does not take mutation ratio into account, it can only emulate BFF's algorithm in configuration selection using a fixed mutation ratio. We note that adding the capability to vary the mutation ratio is prohibitively expensive for us: FuzzSim is an offline simulator, and therefore we need to collect ground-truth data for all possible configurations. Adding a new dimension into our current system would directly multiply our data collection cost.

Going back to our evaluation, let us focus on the Weighted-Random rows in Table 5.2. Density with fixed-run epochs (BFF) yields 84 and 92 bugs in the two datasets. The corresponding numbers for Rate with fixed-time epochs (our recommendation) are 100 and 167, with respective improvements of $1.19\times$ and $1.82\times$ (average $1.5\times$). Based on these numbers, we believe future versions of BFF may benefit from switching over to Rate with fixed-time epochs.

5.10 Discussion

The proposed scheduling algorithms leverage the fact that the outcome of mutational fuzzing can be considered as a bug arrival process with diminishing returns. We also confirmed in our experiment that the speed of bug finding indeed slows down over time, e.g., see Figure 5.4 and 5.5.

However, we notice that if the bug arrival process does not follow a heavy-tailed distribution, then our algorithm can be worse than round-robin. For example, suppose there is a hypothetical configuration that does not return any bugs for 10 hours, and then suddenly returns hundreds of bugs in the next 1 hour. In this case, our algorithm will work poorly because this configuration will not be selected often.

The above scenario is less likely in practice because our model of mutational fuzzing selects test cases from an input space uniformly at random. In order for the above scenario to happen, one needs to select a specific subset of the input space that triggers hundreds of unique bugs only after 10 hours even though we sample the entire space uniformly at random.

5.11 Summary

In this chapter we studied how to find the greatest number of unique bugs in a fuzz campaign. We modeled mutational fuzzing (MBF) as a WCCP process with unknown weights and used the condition in the No Free Lunch theorem to guide us in designing better online algorithms for our problem. In our evaluation of the 26 algorithms presented in this chapter, we found that the fixed-time weighted-random algorithm with the Rate belief metric shows an average of $1.5\times$ improvement over its fixed-run Density-based counterpart, which is currently used by the CERT Basic Fuzzing Framework (BFF). Since our current project does not investigate the effect of varying the mutation ratio, a natural follow-up work would be to investigate how to add this capability to our system in an affordable manner.

Chapter 6

Post-Fuzzing Bug Prioritization

> The superior man makes the difficulty to be overcome his first interest; success only comes later.
>
> —Confucius

An immediate research question that arises after developing several fuzzing strategies is how to prioritize fixing bugs that we found given limited resources, i.e., out of thousands of bugs that we found with fuzzing, which should be fixed first? We noted that not all bugs are equivalent. Some bugs lead to critical security breaches, but some others are just aesthetic bugs that simply annoy users. Thus, we developed a technique, called *automatic exploit generation* (AEG), to prioritize bugs based on the security relevance [14–16, 38].

At a high level, AEG augments typical safety properties in symbolic execution with an exploitability property—which describes the position of attack code, and the value of overwritten addresses, and so forth—and finds a program path where the exploitability property holds. Our analysis is sound, thus, the exploitable test cases generated by AEG lead to a control-hijack attack. Our results on 30 realistic applications on both Windows and Linux showed that AEG is a promising technique for prioritizing software bugs. AEG has several immediate security implications. First, practical AEG fundamentally changes the perceived capabilities of attackers. For example, previously it has been believed that it is relatively difficult for untrained attackers to find novel vulnerabilities and create zero-day exploits. Our research shows this assumption is unfounded. Understanding the capabilities of attackers informs what defenses are appropriate. Second, practical AEG has applications to defense. For example, automated signature generation algorithms

take as input a set of exploits, and output an IDS signature (a.k.a. an input filter) that recognizes subsequent exploits and exploit variants [51, 52]. Automated exploit generation can be fed into signature generation algorithms by defenders without requiring real-life attacks.

6.1 Exploiting Characteristics of Fuzzing Outcome

We observed that not all the bugs found from mutational fuzzing are exploitable. Some bugs are exploitable, but others may not. Therefore, if we can check the exploitability of bugs found, then we can prioritize fixing them: we fix exploitable bugs first because they are critical in security attacks.

Assumption. Our exploitability check does not consider any defense mechanisms such as ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention). We assume that turning a broken exploit that only works under no defense into a weaponized exploit is generally possible with adequate manual effort.

6.2 Problem Statement

At its core, the automatic exploit generation (AEG) challenge is a problem of finding program inputs that result in a desired exploited execution state. In this section, we show how the AEG challenge can be phrased as a formal verification problem, as well as propose a new symbolic execution technique that allows AEG to scale to larger programs than previous techniques. As a result, this formulation: (1) enables formal verification techniques to produce exploits, and (2) allows AEG to directly benefit from any advances in formal verification. We focus on generating a control flow hijack exploit input that intuitively accomplishes two things. First, the exploit should violate safety property, e.g., cause the program to write to out-of-bound memory. Second, the exploit must redirect control flow to the attacker's logic, e.g., by executing injecting shellcode, performing a return-to-libc attack, and others.

At a high level, our approach uses program verification techniques where we verify that the program is exploitable (as opposed to traditional verification that verifies the program is safe). The exploited state is characterized by two Boolean predicates: a buggy execution path formula f_{bug} and a control flow hijack exploit formula f_{exploit}, specifying the control hijack and the code

injection attack. The f_{bug} formula is satisfied when a program violates the semantics of program safety. However, simply violating safety is typically not enough. In addition, f_{exploit} captures the conditions needed to hijack control of the program.

An exploit in our approach is an input ϵ that satisfies the Boolean equation:

$$f_{\text{bug}}(\epsilon) \ \wedge \ f_{\text{exploit}}(\epsilon) \ = \ \text{true} \tag{6.1}$$

Using this formulation, the mechanics of AEG is to check at each step of the execution whether Equation 6.1 is satisfiable. Any satisfying answer is, by construction, a control flow hijack exploit. We discuss these two predicates in more detail below.

The Unsafe Path Formula f_{bug}. f_{bug} represents the path formula of an execution that violates the safety property π. In our implementation, we use popular well-known safety properties for C programs, such as checking for out-of-bounds writes, unsafe format strings, etc. The unsafe path formula f_{bug} partitions the input space into inputs that satisfy the formula (unsafe), and inputs that do not (safe). While path predicates are sufficient to describe bugs at the source-code level, in AEG they are necessary but insufficient to describe the very specific actions we wish to take, e.g., execute shellcode.

The Exploit Formula f_{exploit}. The exploit formula specifies the attacker's logic that the attacker wants to do after hijacking `eip`. For example, if the attacker only wants to crash the program, the formula can be as simple as "set `eip` to an invalid address after we gain control". In our experiments, the attacker's goal is to get a shell. Therefore, f_{exploit} must specify that the shellcode is well-formed in memory, and that `eip` will transfer control to it. The conjunction of the exploit formula (f_{exploit}) will induce constraints on the final solution. If the final constraints (from Equation 6.1) are not met, we consider the bug as non-exploitable.

6.3 AEG: Automatic Exploit Generation

In this section we describe the overall algorithm of AEG. AEG finds exploitable paths with a two-step approach: (1) find a buggy path with traditional symbolic execution, and (2) add additional

exploit formula to check if we can exploit the bug.

Since its introduction in the 70s [26, 88, 94], symbolic execution has been a huge success in software testing [25, 32, 34, 44]. Unlike black-box testing, symbolic execution systematically reasons about programs, making it a white-box testing technique. At a high level, symbolic execution runs a program with a symbolic value as an input, which represents all possible values. As it executes the program under test, it builds symbolic expressions instead of evaluating concrete values. Whenever it reaches a conditional branch instruction, it conceptually forks two symbolic interpreters, one for the true branch and another for the false branch. For every path, a symbolic interpreter builds up a path formula (a.k.a. path predicate) for every branch instruction it encountered during an execution trace. A path formula is satisfiable if there is a concrete input that executes the desired path. One can generate concrete inputs by querying an SMT solver [55] for a solution to a path formula.

Let Symbolic-Execution be a symbolic execution algorithm that takes in a program p and a safety property π, and returns a sequence of path formulas that violates π. Let Exploit-Gen be an exploit generation algorithm that takes in a program and a path formula for a bug found f_{bug}, and outputs an exploitation formula f_{exploit}. Finally, let Exploit-Verify be a verification function that takes in an exploit formula f_{exploit} and a program, and returns either a working exploit if there is a satisfying answer, or $\perp$ if otherwise. Then our AEG algorithm can be described as follows (Algorithm 6.1).

Algorithm 6.1: AEG algorithm.

input : Program p, Safety property π
output: Working exploits $\mathbb{E}$

$\mathbb{E} \leftarrow \varnothing$
for f_{bug} **in** *Symbolic-Execution* (p, π) **do**
  $f_{\text{bug}} \wedge f_{\text{exploit}} \leftarrow$ Exploit-Gen(f_{bug}, p)
  $\epsilon \leftarrow$ Exploit-Verify$(f_{\text{bug}} \wedge f_{\text{exploit}}, \mathrm{p})$
  if $\epsilon \neq \perp$ **then**
    $\mathbb{E} \leftarrow \mathbb{E} \cup \epsilon$
  end
end
return $\mathbb{E}$

6.3.1 Exploit-Gen

Exploit-Gen takes in two inputs to produce an exploit: the unsafe program state containing the path formulas (f_{bug}) and a program under test p to obtain low-level runtime information such as the vulnerable buffer's address, the address of the vulnerable function's return address, and the runtime stack memory contents. If we use concolic testing [71, 137] for generating path formulas, we can directly collect these information without re-running the program under test. Using the information, Exploit-Gen generates exploit formulas ($f_{\text{bug}} \wedge f_{\text{exploit}}$) for four types of exploits: (1) stack-overflow return-to-stack, (2) stack-overflow return-to-libc, (3) format-string return-to-stack, (4) format-string return-to-libc. For brevity, we do not present the full algorithm in this dissertation. Interested readers should refer to our paper [14].

Our current implementation produces two types of exploits: return-to-stack [128] and return-to-libc, both of which are the popular classic control hijack attack techniques. We currently do not handle state-of-the-art protection schemes, but we discuss possible directions in §6.5. Additionally, our return-to-libc attack is different from the classic one in that we do not need to know the address of a "/bin/sh" string in the binary. This technique allows bypassing stack randomization (but not libc randomization).

Return-to-stack Exploit. The return-to-stack exploit overwrites the return address of a function so that the program counter points back to the injected input, e.g., user-provided shellcode. To generate the exploit, AEG finds the address of the vulnerable buffer into which an input string can be copied, and the address where the return address of a vulnerable function is located at. Using the two addresses, AEG calculates the jump target address where the shellcode is located.

Return-to-libc Exploit. In the classic return-to-libc attack, an attacker usually changes the return address to point to the `execve` function in libc. However, to spawn a shell, the attacker must know the address of a "/bin/sh" string in the binary, which is not common in most programs. In our return-to-libc attack, we create a symbolic link to `/bin/sh` and for the link name we use an arbitrary string which resides in libc. For example, a 5 byte string pattern $e8..00...._{16}$ [1] is very common in libc, because it represents a call instruction on x86. Thus, we find a certain string pattern in libc,

[1] A dot (.) represents a 4-bit string in hexadecimal notation.

and generates a symbolic link to `/bin/sh` in the same directory as the target program. The address of the string is passed as the first argument of `execve` (the file to execute), and the address of a null string 00000000_{16} is used for the second and third arguments. The attack is valid only for local attack scenarios, but is more reliable since it bypasses stack address randomization.

Note that the above exploitation techniques (return-to-stack and return-to-libc) determine how to spawn a shell for a control hijack attack, but not how to hijack the control flow. Thus, the above techniques can be applied by different types of control hijack attacks, e.g., format string attacks and stack overflows. For instance, a format string attack can use either of the above techniques to spawn a shell. We currently handle all possible combinations of the above attack-exploit patterns.

6.3.2 Exploit-Verify

Exploit-Verify takes in as input the exploit constraints $f_{\text{bug}} \wedge f_{\text{exploit}}$ and the target binary, and outputs either a concrete working exploit, i.e., an exploit that spawns a shell, or $\perp$, if we fail to generate the exploit. Exploit-Verify first solves the exploit constraints to get a concrete exploit. We use an SMT solver to check satisfiability of the formulas. If the solver finds a satisfying solution then Exploit-Verify moves to the next verification step.

In the second step, Exploit-Verify re-run the target program with the generated exploit to check the exploitability. If the exploit is a local attack, it runs the executable with the exploit as the input and checks if a shell has been spawned. If the exploit is a remote attack, we spawn three processes. The first process runs the executable. The second process runs `nc` to send the exploit to the executable. The third process checks that a remote shell has been spawned at port 31337.

6.3.3 Binary-Only AEG

Binary analysis is more difficult than source-based analysis because binary code lacks high-level abstractions such as types, variables, and functions. Beyond this absence of semantic abstractions, disassembling [96] the binary under test itself, a crucial first step in binary analysis, is challenging. Therefore, binary-level symbolic execution is generally significantly harder than source-based.

However, binary analysis has some advantages over source-based analysis when it comes to security testing, and especially for *AEG*. First, security analysis often does not require source code,

because the analysis also targets commodity software, e.g., Internet Explorer. Second, binary analysis enables us to reason about the low-level details of the code executes: stack-frame layout, calling conventions, etc. These low-level details are important for understanding bugs and crafting exploits. For instance, a buffer overflow exploit requires an attacker to know the exact position of a return address on the stack frame.

Therefore, we implement a binary-level symbolic executor, MAYHEM [38], on top of Binary Analysis Platform (BAP) [28], and use it for automatic bug finding and exploit generation. The results are promising. So far, we have tested 37,391 distinct binaries in Debian, generated 207,206,508 test cases, and found 13,875 unique crashes [16]. Our software testing strategies include not only finding bugs but also prioritizing bugs. *AEG* enables security-based bug prioritization by soundly identifying which bugs are exploitable and should be fixed first.

6.4 Evaluation

6.4.1 Experimental Setup

We evaluated our system on 2 virtual machines running on a desktop with a 3.40GHz Intel(R) Core i7-2600 CPU and 16GB of RAM. Each VM had 4GB RAM and was running Debian Linux (Squeeze) VM and Windows XP SP3 respectively.

6.4.2 Exploitable Bug Detection

We downloaded 29 different vulnerable programs to check the effectiveness of MAYHEM. Table 6.1 summarizes our results. Experiments were performed on stripped unmodified binaries on both Linux and Windows. One of the Windows applications MAYHEM exploited (`Dizzy`) was a packed binary.

Column 3 shows the type of exploits that MAYHEM detected as we described in §6.3.1. Column 4 shows the symbolic sources that we considered for each program. There are examples from all the symbolic input sources that MAYHEM supports, including command-line arguments (Arg.), environment variables (Env. Vars), network packets (Network) and symbolic files (Files). Column 5 is the size of each symbolic input. Column 6 shows the advisory reports for all the demonstrated

	Program	Exploit Type	Input Source	Symbolic Input Size	Advisory ID.	Exploit Gen. Time (s)	Existing Tools
Linux	**A2ps**	Stack Overflow	Env. Vars	550	EDB-ID-816	189	✓
	Aeon	Stack Overflow	Env. Vars	1000	CVE-2005-1019	10	✓
	Aspell	Stack Overflow	Stdin	750	CVE-2004-0548	82	✓
	Atphttpd	Stack Overflow	Network	800	CVE-2000-1816	209	✓
	FreeRadius	Stack Overflow	Env.	9000	Zero-Day	133	✓
	GhostScript	Stack Overflow	Arg.	2000	CVE-2010-2055	18	✓
	Glftpd	Stack Overflow	Arg.	300	OSVDB-ID-16373	4	✓
	Gnugol	Stack Overflow	Env.	3200	Zero-Day	22	**unknown**
	Htget	Stack Overflow	Env. vars	350	N/A	7	**unknown**
	Htpasswd	Stack Overflow	Arg.	400	OSVDB-ID-10068	4	**unknown**
	Iwconfig	Stack Overflow	Arg.	400	CVE-2003-0947	2	✓
	Mbse-bbs	Stack Overflow	Env. vars	4200	CVE-2007-0368	362	✓
	nCompress	Stack Overflow	Arg.	1400	CVE-2001-1413	11	✓
	OrzHttpd	Format String	Network	400	OSVDB-ID-60944	6	**unknown**
	PSUtils	Stack Overflow	Arg.	300	EDB-ID-890	46	✓
	Rsync	Stack Overflow	Env. Vars	100	CVE-2004-2093	8	✓
	SharUtils	Format String	Arg.	300	OSVDB-ID-10255	17	**maybe**
	Socat	Format String	Arg.	600	CVE-2004-1484	47	**maybe**
	Squirrel Mail	Stack Overflow	Arg.	150	CVE-2004-0524	2	✓
	Tipxd	Format String	Arg.	250	OSVDB-ID-12346	10	✓
	xGalaga	Stack Overflow	Env. Vars	300	CVE-2003-0454	3	✓
	Xtokkaetama	Stack Overflow	Arg.	100	OSVDB-ID-2343	10	✓
Windows	**Coolplayer**	Stack Overflow	Files	210	CVE-2008-3408	164	✓
	Destiny	Stack Overflow	Files	2100	OSVDB-ID-53249	963	✓
	Dizzy	Stack Overflow (SEH)	Arg.	519	EDB-ID-15566	13,260	**unknown**
	GAlan	Stack Overflow	Files	1500	OSVDB-ID-60897	831	✓
	GSPlayer	Stack Overflow	Files	400	OSVDB-ID-69006	120	✓
	Muse	Stack Overflow	Files	250	OSVDB-ID-67277	481	✓
	Soritong	Stack Overflow (SEH)	Files	1000	CVE-2009-1643	845	**unknown**

Table 6.1: List of programs that MAYHEM demonstrated as exploitable.

exploits. In fact, MAYHEM found 2 zero-day exploits for two Linux applications, both of which we reported to the developers.

Column 7 contains the exploit generation time for the programs that MAYHEM analyzed. We measured the exploit generation time as the time taken from the start of analysis until the creation of the first working exploit. The time required varies greatly with the complexity of the application and the size of symbolic inputs. The fastest program to exploit was the Linux wireless configuration utility `iwconfig` in 1.90 seconds and the longest was the Windows program `Dizzy`, which took about 4 hours.

The last column shows the comparison against two existing exploitability-testing tools: CERT triage [65] for Linux applications and !exploitable [115] for Windows applications. We ran the tools for both set of applications (Linux and Windows) by feeding crashing inputs found from MAYHEM,

and reported the results in Column 8. When the tool classifies a given crash as exploitable, we denote the result by a check mark (✓). Thus, in this case, there is no difference between MAYHEM and the exiting tools. When the tool returns "unknown", which means the tools cannot determine whether the crash is exploitable, we show "unknown" in the column. Finally, when the tool returns "probably exploitable", we represent the result as "maybe". In our experiment, CERT triage and !exploitable were *unsuccessful* in determining the exploitability of 8 crashes out of 29 total. They classified the 8 crashes as either "unknown" or "maybe". We conclude that MAYHEM can prioritize more number of security bugs than existing tools in our dataset.

6.5 Discussion

Completeness. Our technique is not complete. We do not claim that we can find all possible exploits from a given exploitable bug. It is possible that our analysis misjudges an exploitable bug as not exploitable. However, our analysis is sound. When we say a bug is exploitable, it is indeed an exploitable bug.

Advanced Exploits. We currently focused on stack buffer overflows and format string vulnerabilities. In order to extend MAYHEM to handle heap-based overflows we would likely need to extend the control flow reasoning to also consider heap management structures. Integer overflows are more complicated however, as typically an integer overflow is not problematic by itself. Security-critical problems usually appear when the overflowed integer is used to index or allocate memory. Another potential research direction is to support platform-independent exploitation [37]. We leave adding support for these types of vulnerabilities as future work.

Exploit Variants. Given an exploit formula generated from a bug found, there can be multiple satisfying answers to it. That is, MAYHEM can generate multiple exploits for the same vulnerability: recall from Figure 2.1 that exploitable inputs form a subset of buggy inputs. In our previous study [14] we showed that it is possible to generate hundreds of exploit variants within an hour for a single bug. This result suggests that pattern matching is insufficient to detect malicious threats although existing anti-malware systems still use pattern matching algorithms [36].

Other Exploit Classes. While our definition includes the most popular bugs exploited today, e.g., input validation bugs, such as information disclosure, buffer overflows, heap overflows, and so on, it does not capture all security-critical vulnerabilities. For example, our formulation leaves out-of-scope timing attacks against crypto, which are not readily characterized as safety problems. We leave extending MAYHEM to these types of vulnerabilities as future work.

AEG as a Weapon. One may wonder what if AEG is used as a weapon for attackers. We emphasize that an attacker cannot naively use our framework for real attacks, since AEG does not consider any defense mechanisms. We rather argue that AEG can help defenders in prioritizing fixing bugs by realizing the importance of bugs found. Another potential follow-up question is: could an attacker use the knowledge about AEG to find bugs that AEG would be unlikely to identify as exploitable? We believe that AEG can be used to filter out easy-to-exploit bugs. But, on the other hand, this means the total number of exploitable bugs are reduced for both parties.

Other Bug Finding. In this dissertation, we view AEG as a post-fuzzing process for mutational fuzzing. However, one may ask what if we use other bug finding techniques? Is it possible that other techniques can find more exploitable bugs than mutational fuzzing? Indeed, there is convincing evidence that different bug finding techniques are complementary (see §7.1). In other words, different bug finding may discover distinct sets of bugs for the same program. However, there is no reason to believe that one bug finding technique prevails others in terms of finding exploitable bugs. Furthermore, AEG can be applied in any bug finding technique as a post-fuzzing process.

More on Papers. In this dissertation, we mainly focus on the resource problems in security testing. Therefore, we omit a variety of details and optimizations regarding our system such as preconditioned symbolic execution [14] and symbolic memory optimization [38]. Please refer to our papers for more details about our system.

6.6 Summary

In this chapter, we introduced a novel bug prioritization technique, called AEG. We implemented our approach in a system called MAYHEM and analyzed 29 programs across two different OSes. We

successfully generated 29 control-flow hijack exploits, two of which were against previously unknown vulnerabilities. Furthermore, we compared MAYHEM over two state-of-the-art exploitability checkers—!exploitable on Windows and CERT exploitable on Linux—and showed that MAYHEM can identify 38.1% more exploitable bugs than them.

Chapter 7

Related Work

The work in this dissertation builds upon a very broad base of software testing and verification. Unlike previous works, our focus was on improving the efficiency of mutational fuzzing given limited resources. In this chapter we summarise several related works.

7.1 Bug Finding, Test Case Generation

Since the 70s, researchers have proposed numerous automated methodologies for software testing and test case generation including combinatorial testing [48, 97, 107, 140], constraint-based testing [26, 74, 88, 94, 137], model-based testing [76], random testing [7, 11, 42, 81, 116, 129], and search-based testing [95, 112]. We refer to the latest survey on test case generation for a more complete summary [6], and for symbolic execution [35].

Random Testing and Partition Testing (Subdomain Testing). Partition testing [148] seeks to improve testing performance by dividing the input space of a program into disjoint subsets prior to generating test cases. These subsets constitute equivalence classes of the input space. Then, since all inputs in a subset are equivalent, we need just one test case per subset. Although theoretically attractive, the effectiveness of partition testing largely depends on the quality of the partitioning metrics used in practice. For example, partitioning input space based on code coverage is problematic as it does not produce equivalence classes [79]. Whenever this is the case, the effectiveness of partition testing depends on the distribution of bugs in the input space. For clarity, we use the

term "partition testing" for the case where partitions are disjoint subsets, and "subdomain testing" for the case where partitions may or may not be disjoint (as in [66]). Comparing the effectiveness of subdomain testing with random testing has been an active research area since the late 80s. Hamlet *et al.* [79] and Duran *et al.* [59] showed empirically that random testing can be as effective as subdomain testing in terms of finding bugs. Weyuker *et al.* [147] described analytically when subdomain testing is more likely to find bugs than random testing. Chen *et al.* [40, 41] generalized this result and defined conditions when subdomain testing outperforms than random testing. Follow-up researches [11, 78, 126] also concluded that subdomain testing can always do better than random testing in theory, although random testing performs comparably in practice. This underscores the fact that there is no single best solution for software testing: there is convincing evidence that multiple testing techniques are complementary in terms of finding bugs [106, 121].

Symbolic Execution. Since its introduction in the 70s [26, 88, 94], symbolic execution has been a huge success in software testing [16, 25, 29, 32, 34, 44]. Unlike black-box testing, symbolic execution systematically reasons about programs, making it a white-box testing technique. At a high level, symbolic execution runs a program with a symbolic value as an input, which represents all possible values. As it executes the program under test, it builds symbolic expressions instead of evaluating concrete values. Whenever it reaches a conditional branch instruction, it conceptually forks two symbolic interpreters, one for the true branch and another for the false branch. For every path, a symbolic interpreter builds up a path formula (a.k.a. path predicate) for every branch instruction it encountered during an execution trace. A path formula is satisfiable if there is a concrete input that executes the desired path. One can generate concrete inputs by querying an SMT solver [55] for a solution to a path formula. Dynamic symbolic execution is a variant of traditional symbolic execution, where both symbolic execution and concrete execution operate at the same time. The idea is to use concrete program states to simplify symbolic constraints, e.g., concretizing system calls. We often refer to dynamic symbolic execution as concolic testing (concrete + symbolic) [69, 137].

Fuzzing. Since its introduction in 1990 by Miller *et al.* [116], fuzzing in its various forms has become the most widely-deployed technique for finding bugs. More recently, sophisticated tech-

niques for dynamic test generation have been applied in fuzzing [43]. There are several attempts to utilize evolutionary algorithms in fuzzing [57, 92, 136]. It is also a common practice to employ several heuristics such as code coverage and the distance between instructions [33, 34, 71, 72, 100, 105, 108, 130, 151] to improve the effectiveness of fuzzing.

Combining Testing Techniques. There are several attempts in combining multiple testing techniques. Hybrid concolic testing [106] is the first attempt in combining symbolic execution and mutational fuzzing. It interleaves between mutational fuzzing and concolic testing when there is no more coverage increase with hope that it can discover novel execution paths. Yang *et al.* [153] attempted to use symbolic execution to figure out which input vectors are related. Then they utilized this information to perform combinatorial testing.

7.2 Exploit Generation

Modern AEG research dates back at least to Ganapathy *et al.* [68], who explicitly connected verification to exploit generation. They modeled how format string specifiers are parsed by variadic functions such as `printf`, and used the model to automatically generate exploits. They also demonstrated automatically generating an exploit against a key integrity property for a cryptographic co-processor. However, they only considered API-level exploits, which does not include running shellcode nor the conditions necessary to reach a vulnerable API call site.

In 2007, Medeiros [114] and Grenier *et al.* [75] proposed techniques based on pattern matching for AEG. In 2008, Brumley *et al.* [27] developed automatic *patch-based* exploit generation (APEG). The APEG challenge is: given a buggy program P and a patched version P', generate an exploit for the bug present in P but not present in P'. The idea is that the difference between P and P' reflects (1) where the original bug occurs, and (2) under what conditions it may be triggered. Attackers have long known this, and routinely analyze patches to find non-public bugs. For example, attackers often joke Microsoft's "patch Tuesday" is followed by "exploit Wednesday". Our techniques automatically found the differences between P and P' and generated inputs that triggered the bugs in P using symbolic execution. One main security implication is that attackers can potentially use APEG to exploit bugs before patches can be distributed to a large number of users. We generated

exploits for 5 Microsoft security patches, which included triggering an infinite loop in the TCP/IP driver and stealing files on Microsoft web servers. One limitation was our work only proposed, but did not implement, techniques for executing shellcode for memory safety bugs.

Heelan's thesis work was the first to comprehensively describe and implement techniques for automatically generating control flow hijack exploits that execute shellcode [83]. In Heelan's problem setting, the attacker is given an input that executes an exploitable program path, and the goal is to output a working control flow hijack exploit. This setting is the same as in our running example where we first fuzzed to find bugs, and then checked exploitability. Heelan proposed using symbolic execution and taint analysis to derive the conditions necessary to transfer control to shellcode, and demonstrated a tool that produced exploits for several synthetic and one real vulnerabilities. His work also used a technique called return-to-register to improve exploit robustness. Heelan's thesis also presents a comprehensive history of AEG up through 2009.

Chapter 8

Conclusion

We showed several fuzzing strategies to improve the testing efficiency under constrained resources. This chapter concludes by briefly summarizing our contributions, and then discussing open issues and future work.

8.1 Summary

This dissertation began by formally defining the process of fuzzing that we call fuzz campaign. A fuzz campaign consists of a sequence of fuzz runs that takes in a fuzz configuration—a set of parameters—as input and outputs a stream of log entries that include buggy inputs, bug identifiers and timestamps. The efficiency of fuzzing—the number of bugs found per time—can totally change depending on fuzz configurations used. Therefore, one should carefully choose fuzz configurations in order to maximize the fuzzing efficiency.

Unfortunately, the parameter space for fuzzing is typically too large to be examined exhaustively, thus there are potentially infinite number of fuzz configurations that we can choose for fuzzing. Since we do not have enough resources to test all possible fuzz configurations, we need to reduce the parameter space to consider. In Chapter 3, we investigated several seed reduction strategies that allow an analyst to focus on a small subset of seeds instead of considering potentially infinite number of seeds. Our experimental results confirmed that collecting seeds with good coverage helps improving fuzzing efficiency.

Next, in Chapter 4, we developed a way to infer a "good" mutation ratio for fuzzing directly

from a program execution. Since mutation ratio is a continuous parameter, it can potentially be infinite. We have formulated the failure rate of mutational fuzzing in terms of input-bit dependences in an input. We then designed a new fuzzing framework called SymFuzz that leverages a white-box analysis technique to compute the input-bit dependences in order to estimate a probabilistically optimal mutation ratio for a given program execution.

Although the above techniques help in reducing the number of fuzz configurations to run, we still need to find a way to optimize the time allocation for each of the fuzz configurations. Therefore, we addressed the fuzz configuration scheduling challenge in Chapter 5 that dynamically allocates time for given fuzz configurations in order to maximize the fuzzing efficiency.

Finally, we addressed a post-fuzzing resource problem in Chapter 6. The problem states that we cannot simply fix all the bugs found from fuzzing. To tackle the challenge, we introduced a bug prioritization technique called AEG that identifies the exploitability of a software bug. With AEG, we can prioritize fixing bugs based on the security relevance of them.

8.2 Future Work

Fuzzing. Recall from §2.4, there are many different classes of fuzzing. This dissertation mainly focused on MBF, but MBF traditionally suffers from several issues. First, it is unlikely for MBF to generate absolutely well-formed inputs, e.g., when inputs contain checksum fields. Of course, MBF is still effective in finding security bugs because many security bugs can be triggered with partially-structured inputs. However, considering checksum fields in a seed input would increase the probability of finding buggy inputs. Second, our current model for MBF only generates fixed-size inputs. However, exiting fuzzers can produce mutated test cases that have different size than the seeds. Combining variable-length test case generation algorithms with our approaches may benefit fuzzing, although it is not clear how to model those algorithms formally.

Exploit Generation. Our experiments show that Mayhem can generate exploits for standard vulnerabilities such as stack-based buffer overflows and format strings. An interesting future direction is to extend Mayhem to handle more advanced exploitation techniques such as exploiting use-after-free vulnerabilities and information disclosure attacks. At a high level, it should be possible to de-

tect such attacks using safety properties similar to the ones MAYHEM currently employs. However, it is still an open question how the same techniques can scale and detect such exploits in bigger programs.

Appendices

Appendix A

Proofs

A.1 Solving NLP

Recall from §4.4.2, we solve the NLP problem to optain an optimal mutation ratio for a given minimum buggy bitset.

Theorem 2 (Optimal Mutation Ratio)**.** *Given a minimum buggy bitset B' and the corresponding $\uparrow_s^p(B')$ for a program p and a seed s, let $b = |B'|$ and $d = |\uparrow_s^p(B')|$. The optimal mutation ratio r for finding the bug $\textsc{Triage}(\pi, \sigma_p(\mu(s, B')))$ is*

$$r = \frac{b \times (N+1)}{d \times N} \text{ when } N \cdot r > b. \tag{4.3}$$

Proof. The goal of the NLP is to maximize the following failure rate

$$\theta_b = \frac{\binom{N-d}{N \cdot r - b}}{\binom{N}{N \cdot r}}.$$

For simplicity, we let u to denote $N \cdot r - b$, and v to denote $N - d$. Then the failure rate is simplified as follows.

$$\frac{\binom{v}{u}}{\binom{N}{u+b}}.$$

By expanding the binomial coefficients, we have

$$\frac{\binom{v}{u}}{\binom{N}{u+b}} = \frac{\frac{v!}{u!(v-u)!}}{\frac{N!}{(b+u)!(N-u-b)!}} = \frac{v!(b+u)!(N-b-u)!}{u!(v-u)!N!}.$$

When $u = 1$, the failure rate is

$$\frac{v!(b+1)!(N-b-1)!}{1!(v-1)!N!}.$$

When $u = 2$, the failure rate is

$$\frac{v!(b+2)!(N-b-2)!}{2!(v-2)!N!}.$$

We note that, as we increase u by one, the failure rate increases by the factor of

$$\frac{(v-u+1)(b+u)}{u(N-b-u+1)} \text{ when } u > 0.$$

Since the factor monotonically decreases in terms of u, the maximum failure rate can be achieved when the factor becomes 1. This relaxation gives us the maximum failure rate when

$$\frac{(v-u+1)(b+u)}{u(N-b-u+1)} = 1 \text{ when } u > 0.$$

Solving the equation with respect to u, we have

$$u = \frac{b(v+1)}{N-v} \text{ when } u > 0.$$

Since $u = N \cdot r - b$ and $v = N - d$, we can further simplify the equation with respect to the mutation ratio r:

$$r = \frac{b \times (N+1)}{d \times N} \text{ when } N \cdot r > b.$$

□

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] Humberto Abdelnur, Radu State, Jorge Lucángeli Obes, and Olivier Festor. Spectral Fuzzing: Evaluation & Feedback. Technical Report RR-7193, INRIA Lorraine, 2010.

[3] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computing Surveys*, 14(2):159–192, 1982.

[4] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21 (4):181–185, 1985.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 1 edition, 2008.

[6] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[7] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random Test Run Length and Effectiveness. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 19–28, 2008.

[8] Laurent Andrey, Humberto Abdelnur, Jorge Lucangeli Obes, Olivier Festor, and Radu State. Closed Loop Fuzzing Algorithms. Technical Report ANR-08-VERS-017, Inria, 2010.

[9] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[10] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Formal Analysis of the Effectiveness and Predictability of Random Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 219–229, 2010.

[11] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, 2012.

[12] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The Nonstochastic Multiarmed Bandit Problem. *Journal on Computing*, 32(1):48–77, 2002.

[13] Peter Auer, Nicolò Cesa-Bianchi, and Fischer Paul. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[14] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium*, pages 283–300, 2011.

[15] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2): 74–84, 2014.

[16] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the International Conference on Software Engineering*, pages 1083–1094, 2014.

[17] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[18] Farokh B. Bastani. On the Uncertainty in the Correctness of Computer Programs. *IEEE Transactions on Software Engineering*, 11(9):857–864, 1985.

[19] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[20] Boris Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[21] Jon Bentley and Bob Floyd. Programming Pearls: A Sample of Brilliance. *Communications of the ACM*, 30(9):754–757, 1987.

[22] Donald A Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, 1985.

[23] Dimitri P. Bertsekas and Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 1999.

[24] Yvonne M. Bishop, Stephen E. Fienberg, and Paul W. Holland. *Discrete Multivariate Analysis: Theory and Practice*. MIT Press, Cambridge, 1975.

[25] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the International Conference on Software Engineering*, pages 122–131, 2013.

[26] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[27] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–157, 2008.

[28] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the International Conference on Computer Aided Verification*, pages 463–469, 2011.

[29] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the ACM European Conference on Computer Systems*, pages 183–198, 2011.

[30] Bugcrowd. List of Bug Bounty Programs. https://bugcrowd.com/list-of-bug-bounty-programs/.

[31] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 317–329, 2007.

[32] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[33] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[34] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 209–224, 2008.

[35] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the International Conference on Software Engineering*, pages 1066–1071, 2011.

[36] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. SplitScreen: Enabling Efficient, Distributed Malware Detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 377–390, 2010.

[37] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-Independent Programs. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 547–558, 2010.

[38] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[39] Sang Kil Cha, Maverick Woo, and David Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[40] Tsong Yueh Chen and Yuen Tak Yu. On the Relationship Between Partition and Random Testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[41] Tsong Yueh Chen and Yuen Tak Yu. On the Expected Number of Failures Detected by Subdomain Testing and Random Testing. *IEEE Transactions on Software Engineering*, 22(2): 109–119, 1996.

[42] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive Random Testing: The ART of Test Case Diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[43] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming Compiler Fuzzers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[44] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[45] Vašek Chvátal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[46] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[47] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351, 2005.

[48] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[49] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol Specification Extraction. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

[50] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[51] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 133–147, 2005.

[52] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 117–130, 2007.

[53] NIST Computer Security Resource Center (CSRC). National vulnerability database. http://nvd.nist.gov/home.cfm.

[54] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 391–402, 2008.

[55] Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011.

[56] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[57] Jared D. Demott, Richard J. Enbody, and William F. Punch. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. In *Black Hat USA*, 2007.

[58] Dustin Duran, David Weston, and Matt Miller. Targeted Taint Driven Fuzzing using Software Metrics. In *CanSecWest*, 2011. URL http://goo.gl/sg6H5h.

[59] Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, 1984.

[60] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[61] Michael Eddington. Peach Fuzzing Platform. http://peachfuzzer.com.

[62] Eric Eide and John Regehr. Volatiles Are Miscompiled, and What to Do About It. In *Proceedings of the International Conference on Embedded Software*, pages 255–264, 2008.

[63] Uriel Feige. A Threshold of $\ln n$ for Approximating Set Cover. *Journal of the ACM*, 45(4): 634–652, 1998.

[64] William Feller. *An Introduction to Probability Theory and its Applications*. Wiley, 3rd edition, 1968.

[65] Jonathan Foote. CERT Linux Triage Tools. http://www.cert.org/blogs/certcc/2012/04/cert_triage_tools_10.html.

[66] Phyllis G. Frankl and Elaine J. Weyuker. A Formal Analysis of the Fault-Detecting Ability of Testing Methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, 1993.

[67] Stefan Frei. The Known Unknowns. Technical report, NSS Labs, 2013.

[68] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic Discovery of API-level Exploits. In *Proceedings of the International Conference on Software Engineering*, pages 312–321, 2005.

[69] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[70] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[71] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–166, 2008.

[72] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 55(3):40–44, 2012.

[73] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[74] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62, 1998.

[75] Lurene Grenier and lin0xx. Byakugan: Increase Your Sight. 2007. URL http://aliggo.springnote.com/pages/6938597/attachments/4517781.

[76] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating Finite State Machines from Abstract State Machines. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[77] Oulu University Secure Programming Group. Radamsa. https://code.google.com/p/ouspg/.

[78] Dick Hamlet. When Only Random Testing Will Do. In *Proceedings of the International Workshop on Random Testing*, pages 1–9, 2006.

[79] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[80] Richard W. Hamming. *Coding and Information Theory*. Prentice Hall, 2nd edition, 1986.

[81] K. V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[82] Mark Harman and Phil McMinn. A Theoretical & Empirical Znalysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–83, 2007.

[83] Sean Heelan. *Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities*. Masters thesis, University of Oxford, 2009.

[84] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.

[85] Allen D. Householder. Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File. Technical Report CMU/SEI-2012-TN-018, CERT, 2012.

[86] Allen D. Householder and Jonathan M. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. Technical Report CMU/SEI-2012-TN-019, CERT, 2012.

[87] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[88] William E. Howden. Methodology for the Generation of Program Test Data. *IEEE Transactions on Computers*, C-24(5):554–560, 1975.

[89] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[90] David S. Johnson. Approximation Algorithms for Combinatorial Problems. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 38–49, 1973.

[91] B. D. Jovanovic and P. S. Levy. A Look at the Rule of Three. *The American Statistician*, 51(2): 137–139, 1997.

[92] Roger Lee Seagle Jr. *A Framework for File Format Fuzzing with Genetic Algorithms*. PhD thesis, University of Tennessee, 2012.

[93] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[94] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7): 385–394, 1976.

[95] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[96] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the USENIX Security Symposium*, pages 255–270, 2004.

[97] Rick Kuhn, Raghu Kacker, Yu Lei, and Justin Hunter. Combinatorial Software Testing. *Computer*, 42(8):94–96, 2009.

[98] Caca Labs. Zzuf: Multi-Purpose Fuzzer. `http://caca.zoy.org/wiki/zzuf`.

[99] Launchpad. Bug statistics. `https://bugs.launchpad.net`.

[100] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 19–32, 2013.

[101] Junghee Lim, Thomas Reps, and Ben Liblit. Extracting Output Formats from Executables. In *Proceedings of the Working Conference on Reverse Engineering*, pages 167–178, 2006.

[102] Zhiqiang Lin and Xiangyu Zhang. Deriving Input Syntactic Structure from Execution. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 83–93, 2008.

[103] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach. In *Proceedings of the International Conference on Dependable Systems Networks*, pages 247–256, 2008.

[104] 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 ACM Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[105] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed Symbolic Execution. In *Proceedings of the International Conference on Static Analysis*, pages 95–111, 2011.

[106] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *Proceedings of the International Conference on Software Engineering*, pages 416–426, 2007.

[107] Robert Mandl. Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing. *Communications of the ACM*, 28(10):1054–1058, 1985.

[108] Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Applied Soft Computing*, 5(3):315–331, 2005.

[109] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[110] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[111] William M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1): 100–107, 1998.

[112] Phil McMinn. Search-Based Software Testing: Past, Present and Future. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[113] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: The State of the Art. Technical Report DSTO-TN-1043, Defence Science and Technology Organisation, 2012.

[114] Jason Medeiros. Automated Exploit Development, The Future of Exploitation is Here. Technical report, Grayscale Research, 2007.

[115] Microsoft. !exploitable Crash Analyzer. http://msecdbg.codeplex.com/.

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

[117] Charlie Miller. Fuzz by Number. In *CanSecWest*, 2008. URL https://cansecwest.com/csw08/csw08-miller.pdf.

[118] Charlie Miller. Babysitting an Army of Monkeys. In *CanSecWest*, 2010. URL http://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf.

[119] Charlie Miller and Zachary N. J. Peterson. Analysis of Mutation and Generation-Based Fuzzing. Technical report, Independent Security Evaluators, 2007.

[120] Joan C. Miller and Clifford J. Maloney. Systematic Mistake Analysis of Digital Computer Programs. *Communications of the ACM*, 6(2):58–63, 1963.

[121] David Molnar. *Dynamic Test Generation for Large Binary Programs*. PhD thesis, University of California, Berkeley, 2009.

[122] 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 USENIX Security Symposium*, pages 67–82, 2009.

[123] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[124] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[125] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Computer Science and Applied Mathematics. Academic Press, 2nd edition, 1978.

[126] Simeon Ntafos. On Random and Partition Testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 42–48, 1998.

[127] Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. In *Proceedings of the International Conference on the Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, 1999.

[128] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), 1996.

[129] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering*, pages 75–84, 2007.

[130] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-Data Generation Using Genetic Algorithms. *Journal of Software Testing, Verification And Reliability*, 9(4):263–282, 1999.

[131] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.

[132] Jesse Ruderman. jsfunfuzz. `https://goo.gl/F2iHly`, 2007.

[133] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[134] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information System Security*, 3(1):30–50, 2000.

[135] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[136] Alan C. Schultz, John J. Grefenstette, and Kenneth A. De Jong. Test and Evaluation by Genetic Algorithms. *IEEE Expert*, 8(5):9–14, 1993.

[137] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

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

[139] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[140] Kuo-Chung Tai and Yu Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[141] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[142] Chrome Security Team. Clusterfuzz. `https://code.google.com/p/clusterfuzz/`.

[143] Steven K. Thompson and George A. F. Seber. *Adaptive Sampling*. Wiley, 1996.

[144] Peleus Uhley. A Basic Distributed Fuzzing Framework for FOE. https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html.

[145] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*, pages 86–106, 2012.

[146] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[147] Elaine J. Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[148] Laura J. White and Edward I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–257, 1980.

[149] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[150] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 511–522, 2013.

[151] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proceedings of the International Conference on Dependable Systems Networks*, pages 359–368, 2009.

[152] Bin Xin and Xiangyu Zhang. Efficient Online Detection of Dynamic Control Dependence. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 185–195, 2007.

[153] Jian Yang, Huanguo Zhang, and Jianming Fu. A Fuzzing Framework Based on Symbolic Execution and Combinatorial Testing. In *IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 2076–2080, 2013.

[154] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[155] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 116–127, 2007.

[156] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[157] Michal Zalewski. cross_fuzz. http://goo.gl/2wLLDm, 2011.

[158] Andreas Zeller. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 1–10, 2002.