UNIVERSITY OF HONG KONG

DOCTORAL THESIS

Enhancing Fuzzing Efficacy: an In-Depth Exploration of Different Strategies

Author:
Mingyuan WU

Supervisor:
Prof. Heming CUI
Co-Supervisor:
Prof. Yuqun ZHANG

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Computer Science
Faculty of Engineering

April 12, 2024

Abstract of thesis entitled

Enhancing Fuzzing Efficacy: an In-Depth Exploration of Different Strategies

Submitted by

Mingyuan WU

for the degree of Doctor of Philosophy

at The University of Hong Kong

in April, 2024

In modern society, complicated systems constructed by software play a vital role in most of the existing services and the robustness of such systems has drawn attention from both industry and academia. To ensure the quality of a software system, the efficiency of testing technology is essential since a vulnerability should be eliminated at an early stage. Fuzzing is one of the most efficient testing technologies in software engineering and it has exposed a significant number of real-world vulnerabilities recently. A typical fuzzer randomly generates new input according to its guidance algorithm which is always defined based on expanding code coverage.

While fuzzing has demonstrated its value in everyday use, there is a lack of foundational research in both the academic and industrial sectors regarding various critical mechanisms of fuzzing. This deficiency leaves researchers without the necessary guidance when advancing new research related to fuzzing. Therefore, we determined to conduct comprehensive studies about the fundamental components of fuzzing to obtain more insights for further research.

Specifically, we first evaluate *Havoc* since it is widely adopted in many existing fuzzers as a fundamental fuzzing strategy. We further propose $Havoc_{MAB}$ with a new guidance algorithm according to our findings to schedule mutators automatically from the exploration history. Next, we study the effectiveness of existing gradient-based fuzzers and propose new fuzzers based on new guidance algorithms named *PreFuzz*.

Through our previous research, we also have found that the fundamental guidance mechanism, coverage guidance, can be less effective when fuzzing deep program states of the target programs. Moreover, random fuzzing strategies can explore the target program efficiently, e.g., only adopting *Havoc* can already outperform tons of other approaches significantly. On the contrary, the random fuzzing strategies could be less effective in some scenarios. An application with complicated constraints can easily terminate at the early stage and thus fuzzers with random fuzzing strategies cannot explore it efficiently.

To this end, our further research focuses on addressing two research questions. The first one is whether we can enhance the efficiency of fuzzing by improving coverage guidance itself. The other one is whether there is a way to ensure that fuzzing can explore programs randomly without violating the constraints predefined by the program. To answer these questions, we first propose the concept of *phantom program*, which is built to mitigate the over-compliance of program dependencies to improve the efficiency of coverage guidance. Accordingly, we build a coverage-guided fuzzer namely *MirageFuzz* which performs dual fuzzing for the original program and the phantom program simultaneously and adopts the taint-based mutation mechanism to generate new mutants by combining the resulting seeds from dual fuzzing via taint analysis. The evaluation results show that *MirageFuzz* outperforms the baseline fuzzers from 13.42% to 77.96% in terms of edge coverage averagely in our benchmark.

Secondly, we focus on testing the large-scale, constraint-laden Java Virtual Machine (JVM) to explore how to conduct random exploration without violating constraints. Simultaneously, we also investigate whether there are alternative approaches to coverage guidance when dealing with excessively large target programs. We first propose a coverage-guided fuzzing framework, namely *JITfuzz*, to automatically detect JIT bugs. *JITfuzz* adopts a set of optimization-activating mutators to trigger the usage of typical JIT optimizations, e.g., function inlining and simplification. Meanwhile, given JIT optimizations are closely coupled with program control flows, *JITfuzz* also adopts mutators to enrich the control flows of target programs. Therefore, *JITfuzz* can explore the JIT in JVM randomly without violating constraints. To date, *JITfuzz* detects 36 unknown JVM bugs and 27 of them have been confirmed by the developers. Next, we propose *SJFuzz*, which employs a discrepancy-guided seed scheduler to retain discrepancy-inducing class files and class files that generate discrepancy-inducing mutants for fuzzing guidance. We have reported 46 previously unknown potential issues discovered by *SJFuzz* to the JVM developers where 20 were confirmed as bugs and 16 were fixed.

Our empirical studies have delivered critical insights for future fuzzing research and our proposed techniques inspired by our studies have successfully facilitated fuzzing efficacy. In the future, we plan to improve testing efficacy in more challenging fields, e.g., CPU/GPU testing.

Enhancing Fuzzing Efficacy: an In-Depth Exploration of Different Strategies

by

Mingyuan Wu
B.S. *CUPL* M.S. *SUSTECH*

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy

at

University of Hong Kong
April, 2024

List of Publications

Conferences:

[1] **Mingyuan Wu**, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang, "Enhancing Coverage-guided Fuzzing via Phantom Program," in *the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[2] **Mingyuan Wu**, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang, "SJFuzz: Seed & Mutator Scheduling for JVM Fuzzing," in *the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[3] **Mingyuan Wu**, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang, "JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers," in *the 45th IEEE/ACM International Conference on Software Engineering*, 2023.

[4] **Mingyuan Wu**, HLing Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang, "Evaluating and Improving Neural Program-Smoothing-based Fuzzing" in *the 44th IEEE/ACM International Conference on Software Engineering*, 2022.

[5] **Mingyuan Wu**, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang, "One Fuzzing Strategy to Rule Them All," in *the 44th IEEE/ACM International Conference on Software Engineering*, 2022.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Developing foundational software systems plays a pivotal role in safeguarding our domestic economic security, defense, and other security-relevant domains. In particular, effective software testing and analysis for correctly and rapidly detecting system defects is essential to ensure quality and security when developing complex foundational software systems.

Fuzzing (or fuzz testing) refers to an automated software testing methodology that inputs invalid, unexpected, or random data to programs for exposing unexpected program behaviors (such as crashes, failing assertions, or memory leaks), which can be further inspected or analyzed to detect potential vulnerabilities/bugs [152]. Many existing fuzzers tend to facilitate their vulnerability/bug exposure by optimizing the code coverage of programs under test. Given an initial collection of seeds (i.e., inputs), such coverage-guided fuzzers usually develop strategies to iteratively mutate them to generate new seeds for triggering higher code coverage. Compared to other testing techniques, coverage-guided fuzzing excels in its ability to automatically generate diverse and unexpected inputs, effectively uncovering vulnerabilities and corner cases that might go unnoticed in traditional testing techniques.

Despite the demonstrated effectiveness of fuzzing in exposing vulnerabilities, large research efforts on developing new fuzzers lack insights on the performance impact from the fundamental fuzzing mechanisms. A common practice is to simply propose new fuzzing strategies upon an existing baseline, e.g., AFL [171]. While such strategies sometimes yield good results, it is still challenging to ascertain their effectiveness due to the unknown performance impact from the existing mechanisms of baseline fuzzers. For example, Böhme et al. proposed AFLfast [11] by utilizing a Markov chain model to allocate energy for seed selection based on the original AFL. Although the evaluation results suggest that AFLfast can outperform AFL, the effectiveness of the proposed seed schedule strategy remains uncertain since its performance also relies on applying the *Havoc* mechanism [159] adopted in AFL on a single seed while excluding other strategies.

On the other hand, although many coverage-guided fuzzers effectively enhance

code coverage and bug exposure, their strategies are essentially limited, hindering further performance improvement. Specifically, these strategies often require complete execution on each seed, constrained by strict dependencies between program branches. Consequently, numerous seeds are ineffective in exploring new program states. Even for effective seeds, their iterative executions are still subject to rigorous program dependencies, repeatedly accessing covered states before uncovering new ones. These issues thus compromise the exploration power of the existing coverage-guided fuzzing strategies.

While it is evident that testing specific software systems to ensure their correctness is vital for the correct execution, how to effectively and efficiently test the software systems with complicated constraints (e.g., JVM) remains rather challenging due to the following reasons. First, while random/probabilistic mutation becomes a major paradigm adopted by many fuzzers, it is nevertheless inefficient since massive resulting mutants cannot conform to the complicated constraints, and executing them easily terminates the testing early (e.g., in the input verification phase) to prevent testing deep program states. Second, the insights from existing fuzzers have not been carefully investigated and thus researchers tend to build a new domain-specific fuzzer from scratch [26, 111] without sufficient inspiration from existing tools.

This dissertation aims to address the aforementioned challenges by conducting comprehensive studies to deliver insights and developing new fuzzing strategies to facilitate fuzzing efficacy. Figure 1.1 summarizes the workflow of this dissertation. First, we have conducted two comprehensive studies about the widely-adopted *Havoc* mechanism [159] and neural-program-smoothing-based fuzzing [160]. Subsequently, inspired by the implications derived from the studies, we have enhanced the coverage-guided fuzzing via the proposed phantom program [158] to overcome the limitations of the existing coverage guidance paradigm. Moreover, we have been inspired to adopt the findings on typical foundational software systems to overcome complicated constraints. Specifically for JVMs, our proposed strategies [162, 161] successfully exposed 47 confirmed real-world JVM/JIT bugs. The rest of this dissertation is organized as follows. Chapter 2 introduces the essential background knowledge about fuzzing and the mechanism of Java Virtual Machine (JVM) and its Just-in-time compiler. Chapter 3 and Chapter 4 discuss two typical fuzzing strategies respectively, which are the *Havoc* mechanism [159] and neural-program-smoothing-based fuzzing [160]. Chapter 5 reveals the limitations of traditional coverage guidance strategy and proposes *MirageFuzz* to enhance the fuzzing efficacy by unleashing the potential power of coverage guidance. Chapter 6 and Chapter 7 introduce how we adapt our findings and inspirations to a fundamental fuzzing domain with complicated constraints, i.e., JVM and its JIT compiler.

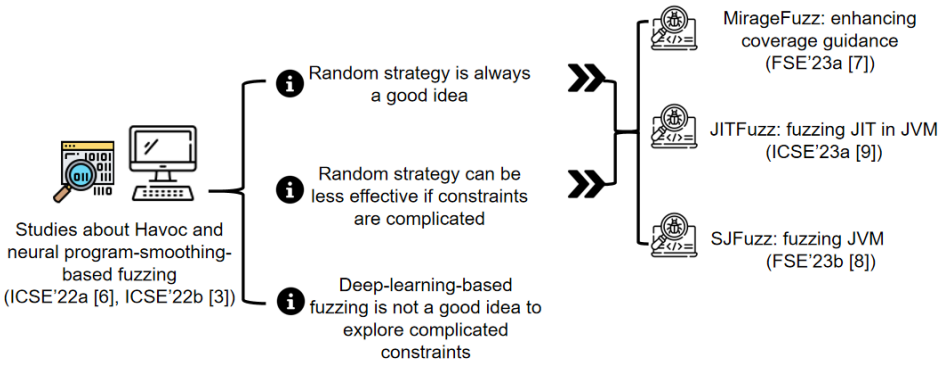


Figure 1.1: The workflow of the dissertation

1.1 Fuzzing Studies

Despite different strategies explained in their papers, many recent coverage-guided fuzzers integrate a lightweight random search mechanism namely *Havoc* to their respective fuzzing strategies without further investigating its rationale or exploring its potential. To investigate its effectiveness for different fuzzers, we have conducted the first comprehensive study of *Havoc* [159] to unleash its potential. The detail of our investigation is presented in Chapter 3. In our evaluation, we have found that executing *Havoc* under one seed without being appended to any fuzzer for sufficient time can already achieve superior edge coverage over many existing fuzzers. Moreover, executing *Havoc* for a longer time upon a fuzzer can potentially result in stronger edge coverage performance. We further infer that the random fuzzing strategy is the key reason to explain why *Havoc* is so effective. The random strategy uniformly selects various searching step lengths to generate seeds, advancing the seeds produced during fuzzing to largely increase the target program coverage across different fuzzing iterations. Inspired by the findings, we further propose $Havoc_{MAB}$, which models the *Havoc* mutation strategy as a multi-armed bandit problem to be solved by dynamically adjusting its mutation strategy. The evaluation result presents that $Havoc_{MAB}$ can significantly increase the edge coverage by 11.1% on average for all the benchmark projects compared with *Havoc*.

Deep learning has demonstrated its effectiveness across numerous domains. Researchers have sought to leverage deep learning to approximate the target programs [131, 130] to facilitate the efficacy of fuzzing by the gradient information guidance from the neural network, namely *neural-program-smoothing-based fuzzing*. However, although they have been shown to be effective in the original papers, it is still unclear how

their key technical components and whether other factors can impact fuzzing performance. To further investigate neural-program-smoothing-based fuzzing, we conducted a study [160] by extensively evaluating the corresponding fuzzers on a large-scale benchmark suite in Chapter 4. In our evaluation, we observe that neural-program-smoothing-based fuzzing leverages neural network models to compute the gradients for reflecting the relations between the program states and seed inputs, i.e., functioning like taint analysis instead of "predicting" the bytes corresponding to unexplored program states. Inspired by such findings, we propose a simplistic technique, *PreFuzz*, which improves neural program-smoothing-based fuzzers with a *resource-efficient edge selection mechanism* to enhance their gradient guidance and a *probabilistic byte selection mechanism* to further boost fuzzing effectiveness. Our evaluation results suggest that *PreFuzz* can significantly outperform the state-of-the-art fuzzer by 43.1% in terms of edge coverage.

From these two studies, we can derive three key insights for further research. Firstly, the significance of employing random strategies stands out crucial in significantly enhancing fuzzing performance without compromising input constraints. Secondly, the utilization of deep learning in the fuzz testing domain requires further exploration; particularly, using it solely for predicting program behavior proves unfeasible. Lastly, there is a pressing need for additional measures to enhance the quality of coverage guidance, thereby further amplifying the efficiency of fuzzing. All these studies have also garnered widespread attention among researchers. The research on the *Havoc* mechanism surprised many researchers with the effectiveness of its random mutation strategy [85], while the recent study on neural-program-smoothing-based fuzzing has sparked intense debates among researchers from different schools of thought regarding its conclusions [128].

1.2 Enhancing Coverage Guidance

Our study about the *Havoc* mechanism has also revealed that fuzzers with random strategy are not capable of exploring program states bounded by rigorous dependencies. We deduce that there are two primary essential challenges regarding coverage guidance. In particular, the adopted seeds of coverage-guided fuzzers are usually ineffective by exploring limited program states since essentially all their executions have to abide by dependencies between program branches while only limited seeds are capable of accessing such dependencies. Moreover, even when iteratively executing such limited seeds, the fuzzers have to repeatedly access the covered program states before uncovering new states. Such facts indicate that exploration power on program states of seeds has not been sufficiently leveraged by the existing coverage-guided fuzzing strategies. To tackle these issues, we propose a coverage-guided fuzzer *MirageFuzz* [158] in Chapter 5, to mitigate the dependencies between program branches when executing seeds for enhancing their exploration power on program states.

Specifically, *MirageFuzz* first creates a "phantom" program of the target program by reducing its dependencies corresponding to conditional statements while retaining their original semantics. Accordingly, *MirageFuzz* performs dual fuzzing, i.e., the *source fuzzing* to fuzz the original program and the *phantom fuzzing* to fuzz the phantom program simultaneously. Then, *MirageFuzz* generates a new seed for the *source fuzzing* via a taint-based mutation mechanism, i.e., updating the target conditional statement of a given seed from the *source fuzzing* with its corresponding condition value derived by the *phantom fuzzing*. The experiment results suggest that *MirageFuzz* outperforms our baseline fuzzers from 13.42% to 77.96% averagely. Furthermore, *MirageFuzz* exposes 29 previously unknown bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

1.3 Fuzzing Foundational Software Systems

Although *MirageFuzz* is effective to fuzz programs with complicated constraints, it can still face a significant challenge if the target program has an additional verification stage whose program state dependencies cannot be easily reduced, e.g., Java Virtual Machine (JVM) and its Just-in-time compiler (JIT). Furthermore, we cannot directly adopt the efficient random strategy such as *Havoc* to fuzz foundational software systems (i.e., JIT and JVM) as it fails to generate seeds satisfying the corresponding constraints. Therefore, the key challenge for fuzzing foundational software systems such as JVM/JIT is how to randomly explore the program states without violating any predefined constraints.

We first proposed a coverage-guided fuzzing framework in Chapter 6, namely *JITfuzz* [161], to automatically detect JIT bugs. In particular, *JITfuzz* adopts a set of optimization-activating mutators to trigger the usage of typical JIT optimizations, e.g., function inlining and simplification. Meanwhile, given JIT optimizations are closely coupled with program control flows, *JITfuzz* adopts mutators to enrich the control flows of target programs. Moreover, *JITfuzz* proposes a random schedule mechanism which we found effective in our *Havoc* study to maximize the code coverage of JIT. Combining all these designs, *JITfuzz* can randomly explore JIT without violating predefined constraints. The experimental results suggest that *JITfuzz* outperforms the state-of-the-art mutation-based and generation-based JVM fuzzers by 27.9% and 18.6% respectively in terms of edge coverage on average. Furthermore, *JITfuzz* also successfully detected 36 previously unknown bugs (including 23 JIT bugs) and 27 bugs (including 18 JIT bugs) have been confirmed by the developers.

To detect the bugs of JVM, prior research work attempts to integrate fuzzing and differential testing for automated JVM testing, i.e., mutating seeds for executing different JVMs such that their discrepant execution results can be used for testing analytics. However, code coverage can hardly be applied to JVM testing techniques because JVMs are likely to cause non-deterministic coverage at runtime. To this end, we propose *SJFuzz* [162] in Chapter 7, the first JVM fuzzing framework with seed and mutator

scheduling mechanisms that schedule seeds on the discrepancy and diversity guidance. The evaluation results show that *SJFuzz* significantly outperforms the state-of-the-art mutation-based and generation-based JVM fuzzers in terms of the inter-JVM discrepancy exposure. Moreover, *SJFuzz* successfully reported 46 potential JVM bugs where 20 have been confirmed as bugs and 16 have been fixed by the JVM developers.

Chapter 2

Background

In this chapter, we give an overview of the essential background knowledge for illustrating the contributions of this dissertation.

2.1 Fuzzing

Fuzzing [153] refers to an automated software testing technique that inputs unexpected or random data to programs such that the program exceptions such as crashes, failing code assertions, or memory leaks can be exposed and monitored. Many existing fuzzing strategies model fuzzing as an optimization problem and attempt to solve it by augmenting code coverage via mutating program seed inputs under a given time budget. Specifically, many fuzzers, e.g., AFL [171], MOPT [94], and Neuzz [131], adopt evolutionary algorithm [165] and utilize code coverage as the fitness function (i.e., guidance) to facilitate bug/vulnerability exposure since it can be advanced in discovering program vulnerabilities without prior program knowledge. In this section, we first introduce the basic framework for evolutionary algorithms and then illustrate how a typical coverage-guided fuzzer integrates evolutionary algorithms.

2.1.1 Evolutionary Algorithm

To solve an optimization problem, an evolutionary algorithm (EA) adopts operations such as mutating the existing solutions to generate new solutions. Among such generated solutions, an EA applies a fitness function to filter them based on their quality such that the remaining ones are retained as one population. Such process is iterated until hitting the preset time budget with the final population returned as the solutions for the optimization problem.

2.1.2 Integrating Fuzzing with EA

Coverage-guided fuzzers often use increased code coverage as the fitness functions. Specifically, they usually adopt edge coverage (where an *edge* refers to a basic-block-wise transition, e.g., a conditional jump in programs) to represent code coverage and retain only the seeds that can trigger new edge coverage for further mutations.

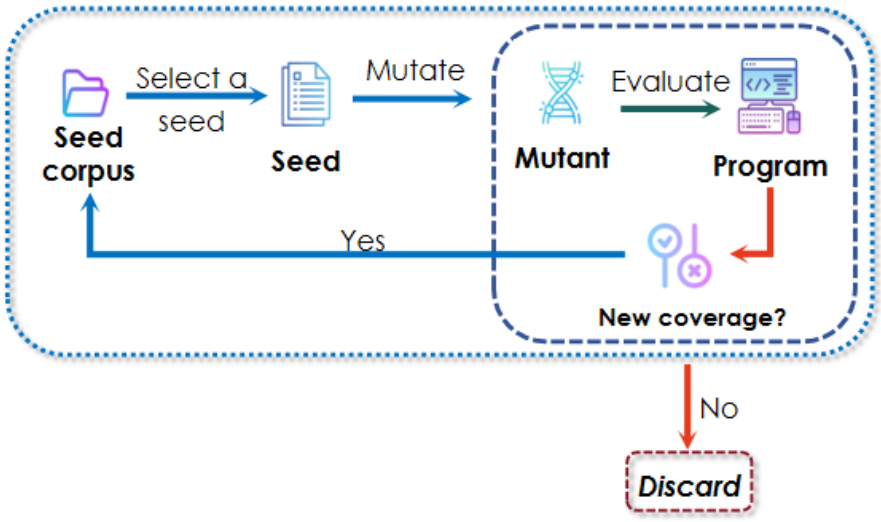


Figure 2.1: A typical coverage-guided fuzzing framework

Figure 2.1 presents a typical coverage-guided fuzzing framework which integrates fuzzing with EA. In particular, a coverage-guided fuzzer first selects a seed (i.e., an input) from the seed corpus, and applies mutation on it to generate a mutant. Subsequently, such a mutant is executed on the target program to explore the program states. If it discovers new code coverage, it will be saved in the seed corpus for further exploration and otherwise will be discarded.

2.2 The Havoc Mechanism

Havoc was first proposed in AFL [171] and later further adopted by many other fuzzers [86, 11, 10, 49, 94]. While their adoptions of *Havoc* can be slightly different, they typically integrate *Havoc* with their major fuzzing strategies (i.e., the core fuzzing strategies) for their iterative executions, i.e., under each iteration, *Havoc* repeatedly mutates each seed provided by (or aggregated to its own seed collection from) executing the major fuzzing strategy via applying multiple randomly selected mutators simultaneously. Figure 2.2 presents the basic workflow of *Havoc*. For each seed in the seed corpus, *Havoc* first determines the count of its mutations based on the real-time seed information, e.g., queuing time of seeds and the existing "interesting" seed number (i.e., the number of the seeds which can explore new edges defined by AFL). Next, each time when mutating a seed, *Havoc* implements mutator stacking, i.e., mutating it by randomly applying multiple mutators (e.g., 15 for AFL, MOPT, etc.) in order from a set of mutators. Note that *Havoc* usually enables a maximum size of such mutator stack (e.g., 128 for AFL, MOPT, etc.) and one mutator can thus be selected multiple times when mutating a given seed. If the generated mutant is "interesting" (i.e., exploring new edges), it will be included as a seed for further mutations. *Havoc* repeats such process until hitting the

mutation count. Accordingly, its fuzzer can resume the execution of its major fuzzing strategy when needed.

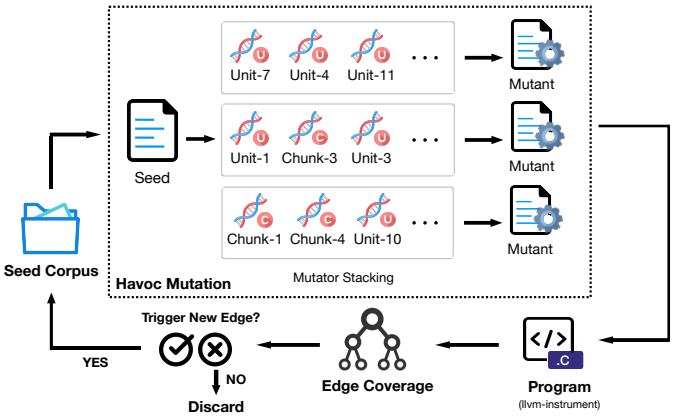


Figure 2.2: The framework of *Havoc*

2.2.1 Mutators and Mutator Stacking

Table 2.1 presents the details of *Havoc* mutators. Note that in the "mutator" column, the number followed by the mutator name refers to the bit-wise mutation range. For instance, `bitflip 1` refers to flipping one random bit at a random position. To our best knowledge, most fuzzers [171, 49, 11, 10, 167, 94] enable a total of 15 mutators for *Havoc*. In this paper, we categorize them into two dimensions: *unit mutators* (labeled in red in Table 2.1) and *chunk mutators* (labeled in blue). In general, *unit mutators* refer to mutating the units of data storage in programs, e.g., bit/byte/word. For example, applying the `bitflip` mutator in Table 2.1 can flip a bit, i.e., switching between 0 and 1. Meanwhile, *chunk mutators* tend to mutate a seed in terms of its randomly chosen chunk. For instance, the `delete bytes` mutator in Table 2.1 first randomly selects a chunk of bytes in the seed and then deletes them altogether.

While many fuzzers mainly apply one mutator to a seed each time, *Havoc* enables mutator stacking to stack and apply multiple mutators on a seed to generate one mutant each time. Typically, *Havoc* first defines a *stacking size* for the applied mutators which is usually randomly determined by the power of two till 128, i.e., 2, 4, 8,...128, for each mutation. Accordingly, *Havoc* can randomly select mutators into the stack where one mutator can be possibly selected multiple times. Eventually, all the stacked mutators are applied to the seed in order to generate a mutant. Note that while most fuzzers uniformly select mutators for their *Havoc*, MOPT and AFL++ adopt a probability distribution generated by Particle Swarm Optimization [75] for *Havoc* to select mutators.

Table 2.1: Mutation operators defined by *Havoc*

Type	Meaning	Mutator
bitflip	Flip a bit at a random position.	`bitflip 1`

interesting values	Set bytes with hard-coded interesting values.	`interest 8`
		`interest 16`
		`interest 32`

arithmetic increase	Perform addition operations.	`addition 8`
		`addition 16`
		`addition 32`

arithmetic decrease	Perform subtraction operations.	`decrease 8`
		`decrease 16`
		`decrease 32`

random value	Randomly set a byte to a random value.	`random byte`
delete bytes	Randomly delete consecutive bytes.	`delete chunk bytes`
clone/insert bytes	Clone bytes in 75%, otherwise insert a block of constant bytes.	`clone/insert chunk bytes`
overwrite bytes	Randomly overwrite the selected consecutive bytes.	`overwrite chunk bytes`

2.2.2 Integration

Havoc can be typically integrated with fuzzers in two manners. One is the *sequential* manner, i.e., appending *Havoc* as a later mutation stage to their major fuzzing strategies. For instance, AFL [171] launches *Havoc* upon the seeds generated after applying its deterministic mutation strategy to generate more seeds under each iterative execution. The other is the *parallel* manner, i.e., applying *Havoc* and the major fuzzing strategy of a fuzzer in parallel. For instance, QSYM [167] enables three threads which execute *Havoc*, AFL deterministic mutation strategy, and concolic execution [55] respectively; more specifically, the first two threads are independently executed in parallel and their respective generated seeds are continuously aggregated to be used for the concolic execution.

While *Havoc* has been widely adopted by the aforementioned fuzzers, it is simply utlized as an implementation option while none of the fuzzers has explicitly explored its potential power, e.g., assessing its mechanism and adjusting its setup. Therefore, our paper attempts to explicitly investigate *Havoc*, i.e., extensively assessing its performance impact to fuzzers and its mechanisms, for better leveraging its power and providing practical guidelines for future research.

2.3 Neural Program-Smoothing-Based Fuzzers

Program smoothing refers to setting up a smooth (i.e., differentiable) surrogate function to approximate program branching behaviors with respect to program inputs [131]. While traditional program smoothing techniques [18, 17] can incur substantial performance overheads due to heavyweight symbolic analysis, integrating such concept with neural network models can be rather powerful since they can be used to cope with high-dimensional optimization tasks, i.e., to resolve (approximate) complex and structured program behaviors. To this end, *Neuzz* [131] and *MTFuzz* [130] are proposed to smooth programs via neural network models and guide mutations by yielding the power of their gradients. Specifically, to formulate the optimization problem for fuzzing, the program branching behaviors are defined as a function $F(x)$, where x represents a seed input in terms of byte sequence and the solution is a vector representing its associated branching behaviors. For instance, a solution vector $[1, 0, 1, ...]$ indicates that the first and the third edges have been accessed/explored while the second one has not. Since $F(x)$ is typically discrete, smoothing programs, i.e., making $F(x)$ differentiable, is essential to cope with the usage of gradients.

We then illustrate the rationale behind *Neuzz* and *MTFuzz*. Note that a program execution path, i.e., a sequence of edges, can be determined by the byte sequence of a seed input. Accordingly, an edge can be accessed/explored when the value of its corresponding bytes satisfies its access condition. Otherwise, one of its "sibling" edges (i.e., edges under one shared prefix edge) can be alternatively accessed. For instance, in Figure 2.3, edge e_0 can be accessed when the value of $seed[i]$ satisfies the access condition for e_0, i.e., $seed[i] < 1$. Hence, mutating such $seed[i]$ can lead to exploring a new branching behavior, i.e., accessing e_0's "sibling" edge e_1 instead of e_0.

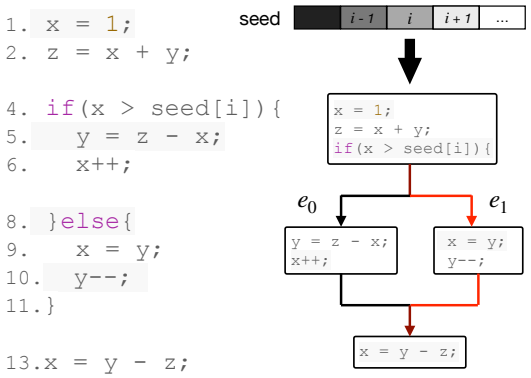


Figure 2.3: An example of neural program-smoothing rationale

Neuzz and *MTFuzz* assume that neural network models can identify the "promising" byte(s) (i.e., the byte(s) corresponding to the access condition) for a previously explored edge. Specifically, the gradient of such byte(s) (e.g., $seed[i]$ in Figure 2.3) to the explored edge is supposed to be larger than other bytes after training. Accordingly,

mutating such byte(s) can indicate that the access condition of the corresponding edge may not be satisfied, i.e., potentially exploring new "sibling" edges. To summarize, *Neuzz* and *MTFuzz* learn to extract the existing branching behaviors to explore new edges rather than predicting "promising" bytes for unseen edges. In particular, their mechanisms commonly consist of two steps: neural program smoothing and gradient-guided mutations as shown in Figure 2.4.

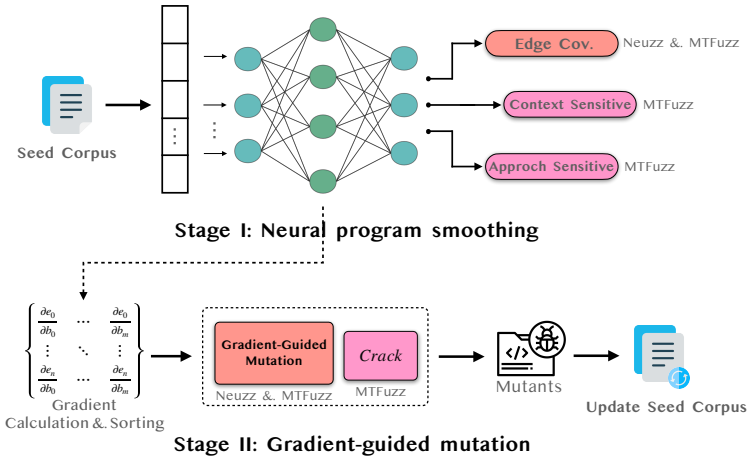


Figure 2.4: Framework of *Neuzz* and *MTFuzz*

2.4 Java Virtual Machine and its Just-in-time Compiler

In this section we give an overview about Java virtual machine (JVM) and its Just-in-time compiler (JIT).

2.4.1 Java Virtual Machine

Java Virtual Machine (JVM) is designed for executing programs which can be compiled to JVM byte code. In general, JVM inputs a class file (a compiled file of a target program), resolves its dependencies, executes the whole program, and returns output. Specifically as shown in Figure 2.5, the `Class Loader` first loads class files into JVM. Next, it sends the instructions to the `Execution Engine` for being executed on the `Operating System`. Subsequently, the `Verification` component verifies whether the instructions are valid or not, and then passes the valid instructions to the `Interpreter`; and aborts the whole execution process otherwise. Note that the instruction could be compiled into the machine code in the `Interpreter` to be executed directly on the corresponding host, which is also known as Just-in-Time compilation [73]. During the entire class file life cycle, the `Garbage Collection` can be activated in any moment to recycle memory for efficient resource management.

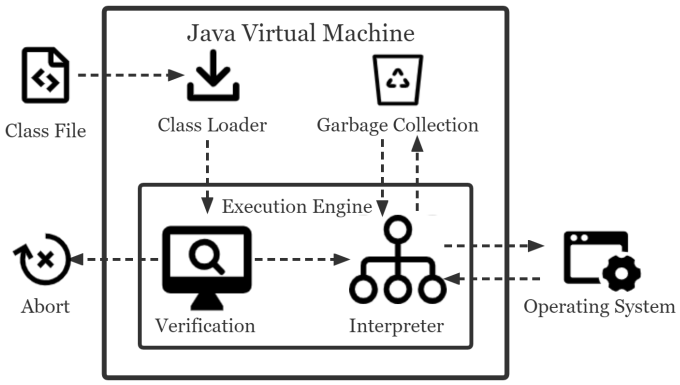


Figure 2.5: The architecture of JVM.

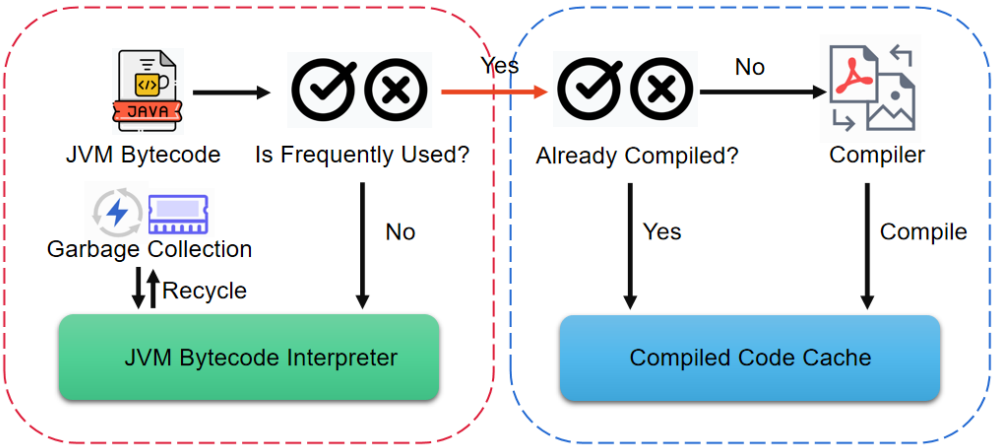


Figure 2.6: The workflow of JVM JIT

2.4.2 JVM JIT

In JVM, the Just-in-time compiler (JIT) selects code (e.g., frequently used methods) running on JVM and compiles them into the native machine code to accelerate the execution for target programs. Figure 2.6 illustrates the workflow of JIT. For a given class file, when JVM executes a method, it first verifies whether such a method is frequently used. If so, the method is then compiled into native code by JIT and stored in the compiled code cache for direct execution. Otherwise it is regularly executed on the JVM bytecode interpreter. Since all such selected methods only need to be compiled once, JIT can significantly advance the execution efficiency of target programs. Note that JIT can also be explicitly activated by specifying the JVM command as `java -Xcomp cls` for a given class `cls`.

JIT adopts multiple optimization techniques [62] to optimize program compilation

performance at runtime for realizing efficient JVM execution via control flow analysis. Specifically, many widely-used JVMs, e.g., OpenJDK [107], OpenJ9 [104], OracleJDK [109], and JRockit [71], adopt common optimization techniques such as function inlining [65], simplification [137], escape analysis [47], and scalar replacement [113]. More specifically, function inlining refers to merging the small-scale functions into their callers to accelerate the frequent function calls. Simplification refers to using an equivalent but simpler expression to replace the given expression for improving runtime efficiency. Escape analysis refers to identifying the dynamic scope of objects and determine whether to allocate them on the Java heap or replace them with constants, i.e., escape. Note that escape analysis can be implemented in multiple levels, including *GlobalEscape* (i.e., objects escape globally), *ArgEscape* (i.e., objects escape within the same thread) and *NoEscape* (i.e., objects do not escape) [30]. Scalar replacement is one typical solution of the escape analysis in the *ArgEscape* level, i.e., replacing objects in the Java heap within the same thread.

Chapter 3

Fuzzing Study: the *Havoc* Mechanism

Notably, many recent coverage-guided fuzzers (e.g., AFL [171], AFL++ [49], MOPT [94], QSYM [167], and FairFuzz [86]) integrate a lightweight random search mechanism namely *Havoc*[1] to their respective fuzzing strategies for increasing their code coverage. For instance, we observe that while the major fuzzing strategy of FairFuzz can explore 12k+ program edges within around 21 hours, its adopted *Havoc* can explore 7.8k+ program edges within only around 3 hours. In contrast to many existing fuzzers which adopt only one mutator under each iterative execution, *Havoc* randomly selects multiple diverse mutators, e.g., flipping a single bit and inserting/deleting a randomly chosen continuous chuck of bytes, and applies them all together for generating one seed during each iteration. Typically, under each iteration, *Havoc* can be integrated with fuzzers either sequentially, i.e., executing *Havoc* upon the seeds collected after executing their major fuzzing strategies, or in parallel, i.e., executing *Havoc* and their major fuzzing strategies at the same time in different processes/threads upon their seed aggregation.

Although *Havoc* has been widely adopted by existing fuzzers, they tend to include *Havoc* only as an implementation option without further investigating its rationale or exploring its potential. For instance, AFL, AFL++ and FairFuzz simply adopt *Havoc* as an additional mutation stage and QSYM utilizes *Havoc* to generate seeds for its concolic execution to increase code coverage. That said, they simply adopt *Havoc* under its default setup, i.e., none of the prior work attempts to study the impact of different *Havoc* settings, explore different ways to integrate *Havoc*, or further boost the *Havoc* strategy itself.

In this chapter, we conduct the first comprehensive study of *Havoc* to unleash its potential. In particular, we first collect 7 recent binary fuzzers and the pure *Havoc* (i.e., applying *Havoc* only without appending it to any fuzzer) as our studied subjects and

[1] While such mechanism may have different names according to different fuzzing papers, we adopt *Havoc* following AFL.

construct a benchmark by collecting their studied projects in common. Then, we conduct an extensive study to investigate how enabling *Havoc* in the studied subjects can impact their performance (e.g., code coverage and bug exposure). Our evaluation results indicate that for all the studied fuzzers, appending *Havoc* to them under its default setup can significantly increase their edge coverage upon all the benchmark projects from 43.9% to 3.7X on average. Meanwhile, we also find that even directly applying the pure *Havoc* only can result in surprisingly strong edge coverage and significantly outperform most of our studied fuzzers. Moreover, while different fuzzers can achieve quite divergent edge coverage results, applying *Havoc* to the studied fuzzers under sufficient execution time can in general not only significantly increase their edge coverage compared with their default *Havoc* integration, but also strongly reduce the performance gap of their edge coverage when applying their original versions. Lastly, *Havoc* can also help all the studied fuzzers expose more unique crashes than their corresponding major fuzzing strategies.

Inspired by our findings, we propose an improved version of *Havoc* namely $Havoc_{MAB}$ which models the *Havoc* mutation strategy as a multi-armed bandit problem (MAB) [155] to be further solved by dynamically adjusting the mutation strategy. The evaluation results indicate that under 24-hour execution, $Havoc_{MAB}$ can outperform the pure *Havoc* significantly by 11.1% in terms of edge coverage on all the benchmarks on average. $Havoc_{MAB}$ can also slightly outperform state-of-the-art QSYM which augments its computing resource by adopting three threads in parallel. Moreover, we also design $Havoc^3_{MAB}$ by executing $Havoc_{MAB}$ with three threads in parallel. The evaluation result indicates that $Havoc^3_{MAB}$ can outperform state-of-the-art QSYM by 9% on average.

3.1 *Havoc* Impact Study

3.1.1 Subjects & Benchmarks

Subjects.

In general, we determine to adopt the following types of fuzzers as our study subjects. First, we attempt to include the fuzzers which originally adopt *Havoc* to expose how *Havoc* can impact their performance by default. Next, we also attempt to explore the fuzzers which do not originally adopt *Havoc* but can possibly integrate *Havoc* under appropriate effort. Accordingly, we can investigate whether and how *Havoc* can be effective in a wider range of fuzzers. At last, we also include the pure *Havoc*, i.e., using only one seed to launch *Havoc* for generating new seeds without appending it to any fuzzer, for analyzing how the power of *Havoc* can be unleashed.

Note that while there are many existing fuzzers which can meet our selection criteria above, we also need to filter them for selecting the representative ones. To this end, we first determine to limit our search scope within the fuzzers published in the top software engineering and security conferences, i.e., ICSE, FSE, ASE, ISSTA, CCS,

S&P, USENIX Security, and NDSS, of recent years. Furthermore, we can only evaluate the fuzzers when their source code are fully available and can be successfully executed. At last, it is rather challenging to integrate *Havoc* with certain potential fuzzers due to the engineering-/concept-wise challenges. Therefore in this study, we only target AFL variants due to the appropriate workloads for implementing *Havoc* for them.

Eventually, we select 8 representative fuzzers as our studied subjects, including 5 fuzzers with *Havoc* (AFL [171], AFL++ [49], MOPT [94], FairFuzz [86], QSYM [167]), 2 fuzzers without *Havoc* (*Neuzz* [131], *MTFuzz* [130]) and the pure *Havoc* itself. Note that such subjects can be rather representative in terms of technical designs, i.e., including AFL-based, concolic-execution-based, and neural program-smoothing-based fuzzers.

Benchmark programs.

We construct our benchmark based on the projects commonly adopted by the original papers of our studied fuzzers [86, 94, 167, 131, 130]. In particular, we select 12 frequently used projects out of the papers to form our benchmark for evaluation. More specifically, we first select all 6 projects that are adopted by at least 3 papers; then, we further randomly select another 6 projects which are adopted by one or two papers. The selection details are presented in our Github page [124].

3.1.2 Evaluation Setups

Our evaluations are performed on ESC servers with 128-core 2.6 GHz AMD EPYC™ ROME 7H12 CPUs and 256 GiB RAM. The servers run on Linux 4.15.0-147-generic Ubuntu 18.04. The evaluations that involve deep learning model training (i.e., *Neuzz* and *MTFuzz*) are executed with four RTX 2080ti GPUs.

We strictly follow the respective original procedures of the studied fuzzers to execute them.Specifically, we set the overall execution time budget for each fuzzer 24 hours following prior works [76, 10, 131, 11, 86, 130]. Note that we run each experiment five times for obtaining the average results to reduce the impact of randomness. Notably, all the studied fuzzers are executed with the programs based on AFL instrumentation to collect the runtime coverage information. To this end, we apply the AFL (v2.57) llvm-mode (llvm-8.0) to instrument the source code during compilation. We also follow the instructions mentioned in previous work [76, 64, 86, 150, 25] to construct the initial seed corpus. In particular, we collect initial seeds for `libjpeg`, `libtiff` and `jhead` from AFL official seed corpus [170] and for the rest projects from their own test suites.

We adopt the edge coverage to reflect code coverage where an edge refers to a transition between program blocks, e.g., a conditional jump. We then measure it via the edge number derived by the AFL built-in tool named `afl-showmap`, which has been widely used as a guidance function by many existing fuzzers [86, 167, 130, 131, 21]. Note that the AFL authors also refer to such metrics as "a better predictor of how the tool will fare in the wild" [169].

Table 3.1: The edge coverage performance of fuzzers with *Havoc*

Programs	AFL		AFL++		FairFuzz		MOPT		QSYM	
	Original	Major	Original	Major	Original	Major	Original	Major	Original	Major
readelf	12,112	11,598	9,844	8,268	36,372	20,184	67,505	8,006	69,597	15,984
nm	7,188	6,004	6,049	5,563	16,456	10,627	23,159	5,413	30,359	8,024
objdump	13,748	13,154	13,473	11,829	24,204	16,246	38,027	11,698	41,097	13,439
size	8,262	8,904	4,205	3,643	16,547	10,503	19,172	3,593	23,700	8,459
strip	15,310	14,938	12,116	9,425	26,622	20,870	37,318	9,814	46,633	16,287
djpeg	5,388	7,756	5,474	3,174	10,405	7,782	15,805	3,222	19,295	7,980
tcpdump	14,972	7,192	5,294	1,758	18,457	11,130	44,275	2,477	45,804	16,385
xmllint	18,189	15,314	15,986	13,955	28,174	14,817	49,618	13,632	46,538	20,635
tiff2bw	5,372	5,658	3,767	3,127	8,834	6,340	8,707	3,148	9,301	6,984
mutool	11,529	10,064	11,028	5,677	14,430	10,397	17,438	5,867	17,827	13,653
harfbuzz	21,713	20,513	16,411	9,651	29,939	27,262	54,178	10,365	59,270	26,574
jhead	1,012	636	1,057	404	1,114	633	1,127	452	4,516	2,047
Average	11,233	10,144	8,725	6,373	19,269	13,066	31,361	6,474	34,495	13,038

Table 3.2: The impact of *Havoc* on *Neuzz* and *MTFuzz*

Programs	Pure Havoc	*Neuzz*			*MTFuzz*		
		Origin	Integration		Origin	Integration	
			Havoc	Major		Havoc	Major
readelf	73,478	43,040	18,530	27,699	40,594	13,967	31,220
nm	20,696	16,002	8,617	12,469	20,863	9,841	12,745
objdump	37,401	29,155	14,619	18,661	25,369	12,057	18,784
size	17,634	13,228	6,191	8,040	12,256	8,593	6,686
strip	38,200	29,767	16,117	18,959	28,981	14,098	22,884
djpeg	16,142	15,805	12,861	8,549	7,640	7,432	5,142
tcpdump	43,482	17,216	20,704	5,755	14,067	19,237	9,727
xmllint	49,269	28,213	15,891	21,386	27,682	14,228	24,692
tiff2bw	8,516	9,168	3,174	6,016	7,254	2,088	5,806
mutool	17,014	15,560	5,171	13,196	14,391	3,976	12,961
harfbuzz	50,549	38,726	12,548	30,191	41,691	15,203	33,742
jhead	1,132	1,078	705	407	992	698	400
Average	31,126	21,413	11,261	14,277	20,148	10,118	15,399

3.1.3 Research Questions

We investigate the following research questions for extensively studying *Havoc*.

- **RQ1**: How does the default *Havoc*, i.e., the direct application of *Havoc* without modifying its setup or mechanism, perform on different fuzzers? For this RQ, we attempt to investigate the performance impact of the default *Havoc* used in the studied fuzzers.
- **RQ2**: How does *Havoc* perform on different fuzzers under diverse setups? For this RQ, we investigate the performance impact of *Havoc* by enabling *Havoc* in the studied subjects under different execution time setups.

3.1.4 Result Analysis

RQ1: performance impact of the default *Havoc*

We first investigate the impact of the default *Havoc* on the fuzzers with *Havoc*. There can be typically two default setting types for integrating *Havoc* to fuzzers. For many fuzzers which append *Havoc* as a later fuzzing strategy to their major fuzzing strategies under each iterative execution, *Havoc* is launched upon the termination of their major fuzzing strategies and terminated after hitting the mutation count determined at runtime without any specific execution time control by default. As a result, we can infer that the execution time of the default *Havoc* cannot be deterministic. On the other

hand, for the fuzzers which execute *Havoc* and their major fuzzing strategies in parallel, the default *Havoc* is usually executed all along under the execution time. Therefore, its execution time can be typically equal to the overall execution time.

Figure 3.1 presents the execution time distribution of all the studied fuzzers under the total execution time 24 hours (note that *Neuzz* and *MTFuzz*, marked in red, do not have the *Havoc* stage by default, and will be discussed later). We can observe that while AFL, AFL++, and FairFuzz allow quite limited total execution time of *Havoc* by default (i.e., from 0.79 hour to 3.09 hours), MOPT and QSYM allow much longer execution time for *Havoc*. Note that the default setting of QSYM utilizes three threads including the default *Havoc*. Thus *Havoc* is executed in QSYM for the whole 24 hours.

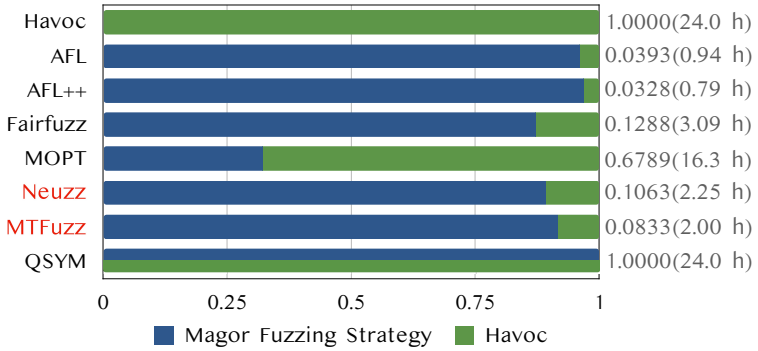


Figure 3.1: Execution time distribution within 24 hours

We first study the *Havoc* impact on the five fuzzers with *Havoc*. Specifically, we create their variants by deleting *Havoc* from their original implementations, i.e., only retaining their major fuzzing strategies. Table 3.1 presents the edge coverage results of the five fuzzers with *Havoc* in terms of their major fuzzing strategies (represented as "Major") and the original implementations (represented as "Original") respectively. Generally, we can observe that the edge coverage of all the studied fuzzers decreases significantly after deleting *Havoc* from their implementations averagely, i.e., 9.7% in AFL, 27.0% in AFL++, 32.2% in FairFuzz, 79.4% in MOPT, and 62.2% in QSYM. Combining Figure 3.1, we can further infer that the 79.4% edge coverage decrease for MOPT is caused by reassigning 16.3 hours (67.9% of all time budget) originally spent on *Havoc* to its major strategy; the 62.2% edge coverage decrease of QSYM is caused by excluding the thread executing *Havoc*. Even for AFL and AFL++ which executes their *Havoc* only less than 1 hour, excluding *Havoc* decreases 9.7% in AFL and 27.0% in AFL++ in terms of edge coverage. All such facts indicate that *Havoc* can significantly increase the edge coverage over the major fuzzing strategies.

We also attempt to append the default *Havoc* into the fuzzers without *Havoc*, i.e., *Neuzz* and *MTFuzz*, and further investigate how the default *Havoc* can impact their edge coverage performance. Specifically, their integration follows the sequential pattern adopted by many existing fuzzers, i.e., appending *Havoc* after executing the original fuzzing strategies of *Neuzz* and *MTFuzz* under each iterative execution. Therefore,

the execution time of the default *Havoc* adopted by them cannot be deterministic. In particular, their execution time distributions are presented in Figure 3.1 where *Neuzz* spends 2.25 hours and *MTFuzz* spends 2 hours on executing the default *Havoc*.

Table 3.2 presents the edge coverage results where "Origin" refers to the original versions of *Neuzz* and *MTFuzz*, while "Integration" refers to *Neuzz* and *MTFuzz* integrated with *Havoc*. We can observe that overall, for the new integrated version, *Havoc* can achieve 78.9%/66.0% higher edge coverage than the major fuzzing strategy of *Neuzz*/*MTFuzz* on average. Moreover, the integrated fuzzers can achieve rather significant performance gain, i.e., 19.3% over the original *Neuzz* and 26.6% over the original *MTFuzz*. To summarize, we can derive that for all the studied fuzzers (no matter originally integrated with *Havoc* or not), appending the default *Havoc* to them can significantly enhance their major/original fuzzing strategies.

> *Finding 1: Applying Havoc by the default setup can significantly improve the edge coverage performance of the studied fuzzers.*

Interestingly, we can find from Table 3.2 that the pure *Havoc*, i.e., using only one seed to launch *Havoc* and executing it all along without appending it to any fuzzer, preforms rather strong in terms of edge coverage, i.e., 31K+ edges on average on all the benchmark projects. More specifically, the pure *Havoc* can significantly outperform most of the studied fuzzers, e.g., 177% over AFL, 257% over AFL++, 45% over *Neuzz*, while obtaining close performance with MOPT and QSYM. Note that while we can definitely enable multiple ways, e.g., applying more than one seed, to launch the execution of the pure *Havoc*, the fact that using one seed can already achieve such superior performance can be a strong evidence that *Havoc* itself is a powerful fuzzer.

> *Finding 2: Havoc is essentially a powerful fuzzer—executing Havoc under one seed without being appended to any fuzzer for sufficient time can already achieve superior edge coverage over many existing fuzzers.*

We then investigate the correlation between the edge coverage performance and the execution time of *Havoc*. We can observe that while *MTFuzz*, QSYM, and the pure *Havoc* can achieve much stronger edge coverage over the other fuzzers according to Tables 3.1 and 3.2, they also have longer execution time for *Havoc* as shown in Figure 3.1. More specifically, the ranking of the edge coverage performance can almost strictly align with the ranking of the execution time of *Havoc* among all the studied fuzzers (except for *Neuzz* and FairFuzz). Therefore, we can infer that for most fuzzers, executing *Havoc* for longer time potentially results in higher edge coverage.

> *Finding 3: Executing Havoc for a longer time upon a fuzzer can potentially result in stronger edge coverage performance.*

Table 3.3: Edge coverage results of fuzzers with modified *Havoc*

Programs	Havoc	QSYM	AFL		AFL++		FairFuzz		MOPT		*Neuzz*		*MTFuzz*	
			Orig	New	Orig	New	Orig	New	Orig	New	Orig	New	Orig	New
readelf	73,478	69,597	12,112	73,842	9,844	72,766	36,372	71,689	67,505	73,175	43,040	70,358	40,594	69,824
nm	20,696	30,359	7,188	21,398	6,049	25,259	16,456	21,537	23,159	26,602	16,002	22,258	20,863	24,387
objdump	37,401	41,097	13,748	36,775	13,473	35,004	24,204	35,802	38,027	37,358	29,155	35,739	25,369	36,203
size	17,634	23,700	8,262	17,296	4,205	18,393	16,547	18,118	19,172	18,707	13,228	16,121	12,256	17,395
strip	38,200	46,633	15,310	37,136	12,116	37,419	26,622	37,724	37,318	40,006	29,767	35,147	28,981	37,548
djpeg	16,142	19,295	5,388	18,543	5,474	15,628	10,405	14,660	15,805	18,127	15,805	23,420	7,640	15,962
tcpdump	43,482	45,804	14,972	40,581	5,294	41,178	18,457	40,407	44,275	44,394	17,216	39,687	14,067	42,317
xmllint	49,269	46,538	18,189	45,869	15,986	46,379	28,174	45,004	49,618	47,190	28,213	45,985	27,682	47,365
tiff2bw	8,516	9,301	5,372	8,093	3,767	7,645	8,834	8,204	8,707	8,083	9,168	9,260	7,254	8,671
mutool	17,014	17,827	11,529	17,325	11,028	17,280	14,430	17,065	17,438	17,504	15,560	19,438	14,391	18,554
harfbuzz	50,549	59,270	21,713	56,058	16,411	52,451	29,939	50,619	54,178	59,314	38,726	51,498	41,691	52,964
jhead	1,132	4,516	1,012	1,129	1,057	1,124	1,114	1,123	1,127	1,133	1,078	1,127	992	1,134
Average	31,126	34,495	11,233	31,170	8,725	30,877	19,296	30,163	31,361	32,633	21,413	30,836	20,148	31,027

RQ2: performance impact of *Havoc* under diverse setups

Table 3.4: Average edge coverage results under different execution time setups

Setup		AFL_{havoc}	AFL++_{havoc}	FairFuzz_{havoc}	MOPT_{havoc}	$\textit{Neuzz}_{havoc}$	$\textit{MTFuzz}_{havoc}$
Total	**Iteration**						
24h	1h	31,170	30,877	30,163	32,633	30,836	31,027
	2h	30,451	31,069	30,853	31,465	31,259	31,265
	4h	30,315	30,541	31,296	32,354	31,462	31,472
	12h	30,567	30,247	30,543	31,764	30,975	30,865

Inspired by the previous findings, we attempt to further investigate the performance impact of *Havoc* on the fuzzers under diverse execution time setups. Specifically, while implementing the default *Havoc* does not concern its execution time, executing *Havoc* under diverse execution time setups essentially demands the modified implementation of integrating *Havoc* to the fuzzers (i.e., the modified *Havoc*).

Implementation. Note that in this paper, we first modify the implementation for integrating *Havoc* to fuzzers in the sequential manner. To begin with, it is essential to figure out how to control the execution time of the major fuzzing strategy and *Havoc* of a fuzzer. Specifically, our insight is to retain the fuzzing states of the major fuzzing strategy and *Havoc* when they are halting. To this end, while realizing such insight by directly integrating the source code of *Havoc* into different fuzzers essentially demands substantial engineering effort, we decide to adopt *socket* programming [156] as an alternative solution, which can execute the major fuzzing mechanism and its appended *Havoc* in different processes since its built-in blocking mechanism can provide the "wake up" function for both monitoring the execution time of an event given its preset timeout and retaining the fuzzing states while halting. Note that such a solution can be quite consistent with a single-process fuzzer in terms of CPU resource consumption. Specifically in the beginning, we execute the major fuzzing strategy of a fuzzer for time duration t to generate new seeds. Subsequently, we transmit the file names of the generated seeds to *Havoc* by *socket*. After completing the whole seed transmission, *Havoc* is executed for time duration t as well while the execution of the original fuzzing strategy is paused. Note that instead of dynamically setting a mutation count for controlling its execution as the default *Havoc*, our modified *Havoc* iteratively generates new seeds based on the updated collection of the "interesting" seeds within time duration t. Similarly after executing *Havoc*, we transmit the file names of its generated seeds to

the original fuzzing strategy of the fuzzer via *socket* for further seed generations. Such a process is iterated until hitting the total time budget.

Evaluation. We first evaluate *Havoc* by setting the iterative time duration t of the major fuzzing strategy/*Havoc* as 1 hour (i.e., executing them for 1 hour respectively under each iteration). As a result, for each fuzzer, its modified *Havoc* can be executed within a total of 12 hours under our 24-hour budget. Table 3.3 presents the evaluation results of the fuzzers with and without applying such modified *Havoc* where "Orig" represents the original fuzzers with their default implementation and "New" represents the associated fuzzer integrated with the modified *Havoc*. Note that since such setup does not fit for the essential mechanisms of the pure *Havoc* and QSYM which execute *Havoc* for the whole execution, i.e., 24 hours, we retain their results of the previous evaluations in Table 3.3 simply for illustration and comparison.

We can observe that while MOPT with the modified *Havoc* can incur quite close edge coverage compared with its default *Havoc* integration as in Table 3.1, the rest fuzzers with the modified *Havoc* can achieve much higher edge coverage compared with their original versions, e.g., 1.8X for AFL. Such a result can further validate our Finding 3. Specifically, the original MOPT can already incur quite a long execution time for *Havoc* by default, i.e., 16.3 hours, and thus can result in rather a strong edge coverage. On the other hand, the execution time of *Havoc* for the other fuzzers turns to be much longer with our new hybrid strategy and thus results in a significant performance gain. Note that for the fuzzers which originally adopts no *Havoc* (i.e., *Neuzz* and *MTFuzz*), their edge coverage performance can also be significantly improved compared with their original versions.

More interestingly, we can find that for most fuzzers, they can incur quite close edge coverage with the modified *Havoc* on all the benchmark projects averagely, i.e., around 31K. Moreover, their project-wise performance can be quite close as well, e.g., around 71K in project `readelf` and 38K in project `objdump`. Compared with the edge coverage from their original versions, their performance gaps are significantly reduced. To illustrate, we adopt the STD (Standard Deviation) of the average edge coverage for the studied fuzzers. Specifically, the STD of all the fuzzers with our new strategy for integrating *Havoc* is 819 compared with that of 8,879 when using their default strategies for integrating *Havoc*, while their average edge coverage is 31,118 compared with 20,278. Such result can indicate that by executing *Havoc* for sufficient time, the edge coverage performance gaps of different fuzzers can be significantly reduced. On the other hand, while the performance of many studied fuzzers are significantly improved by extending the execution time of *Havoc* in a sequential manner, their performance is rather close to the pure *Havoc*. Such facts indicate that *Havoc* can potentially dominate many fuzzers in terms of edge coverage.

We also include block coverage rate (i.e., the number of accessed basic blocks divided by the total number of basic blocks) [167, 90] to strengthen our findings. We can

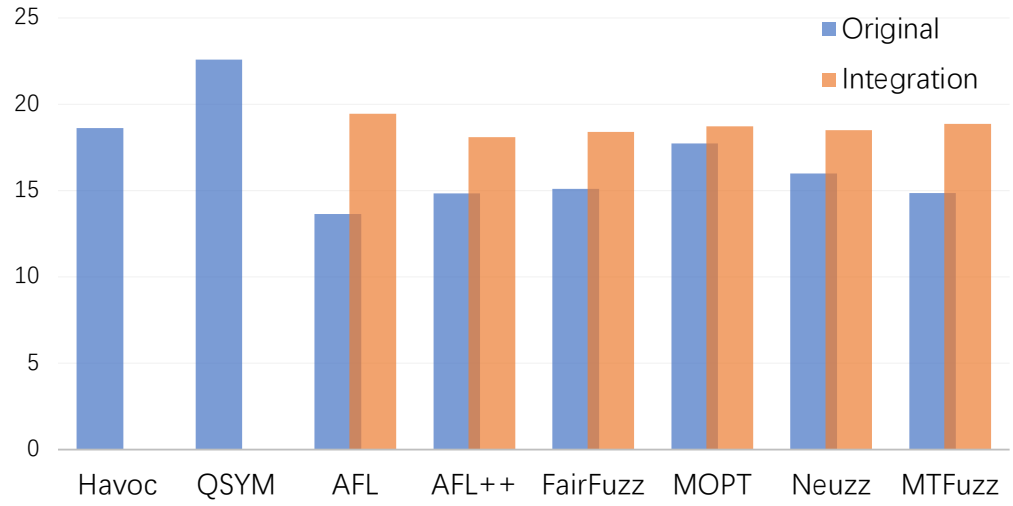


Figure 3.2: Block coverage of different fuzzers with modified *Havoc*

observe in Figure 3.2 that almost all the studied fuzzers significantly increase block coverage compared with their original implementations, e.g., 42.7% for AFL, and 26.9% for *MTFuzz*. Moreover, all the studied fuzzers achieve quite close block coverage rates after integrating *Havoc* with the average block coverage rate of 18.7% and STD of 0.00465, which are consistent with the edge coverage trends.

> *Finding 4: Executing Havoc for sufficient time can dominate the performance of the studied fuzzers and significantly reduce their performance gaps.*

We further attempt to investigate how changing the integration mode of *Havoc* with fuzzers can impact their edge coverage performance. To this end, we first enable diverse setups of the iterative time duration t of *Havoc* in terms of 2 hours, 4 hours, and 12 hours under the total execution time of 24 hours. Table 3.4 presents the evaluation results under such setups. We can observe that overall, there is no significant performance difference under all the setups. Specifically, the largest gap of the average edge coverage of a given fuzzer is only 3.76%. Such a fact can indicate that the edge coverage performance is somewhat resilient to time duration t, i.e., under sufficient total execution time, adapting the execution time of *Havoc* under each iteration results in a rather limited impact on the edge coverage of the associated fuzzer.

> *Finding 5: As long as the total execution time of Havoc is fixed, how to adapt its iterative execution can have a limited impact on the edge coverage performance of the associated fuzzer.*

While the performance gaps between different fuzzers can be significantly reduced by applying the modified *Havoc*, QSYM can still outperform the other fuzzers by at least 10%. Accordingly, we hypothesize that executing multiple fuzzing strategies in

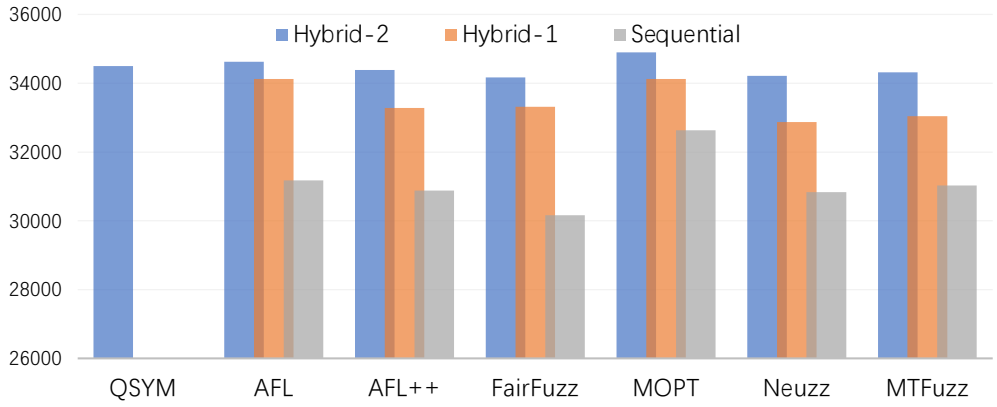


Figure 3.3: Edge coverage of different fuzzers with the hybrid integration of *Havoc*

parallel can be potentially more advanced in boosting edge exploration. We then attempt to validate such hypothesis by also adopting additional threads for executing *Havoc* in parallel in our studied fuzzers, i.e., while retaining the execution of their modified *Havoc* in a sequential manner for 12 hours, we also execute *Havoc* for the whole 24 hours in parallel in additional threads. In particular, we adopt one and two additional threads for executing *Havoc* respectively. Figure 3.3 presents our evaluation results. We observe that when adopting one additional thread to execute *Havoc* (labeled as "Hybrid-1"), averagely the edge coverage of all the studied fuzzers can be increased by 7.4%. Moreover, we can also observe that compared with adopting one additional thread for *Havoc*, adopting two additional threads for *Havoc* (labeled as "Hybrid-2") can further increase the edge coverage performance by 2.9% on top of all the studied fuzzers. Compared with QSYM which originally achieves the optimal performance via using three threads for fuzzing, MOPT and AFL can now even incur performance gains of 1.2% and 0.4%. On the other hand, since the performance gain by simply increasing additional threads for executing *Havoc* becomes marginal, we can infer that simply investing more computing resources on executing *Havoc* may not be cost-effective.

> *Finding 6: Investing more computing resource in executing Havoc can potentially reduce its execution time for approaching the performance bound, but may not be cost-effective.*

While the previous findings reveal that under sufficient execution time of *Havoc*, multiple fuzzers can approach quite close edge coverage performance, we further attempt to investigate how common their explored edges can be. To this end, we determine to adopt the concept of *Jaccard Distance* [154] to delineate the similarity of the explored edges from different fuzzers. In particular, *Jaccard Distance* is usually used to measure the dissimilarity between two sets by dividing the difference of their union size and intersection size by their union size. Figure 3.4 presents the evaluation results of *seed dissimilarity* between the pure *Havoc* and the other fuzzers (with the modified

Havoc) on average, ranging from 0.134 to 0.256. Such a result indicates that applying *Havoc* to different fuzzers can potentially explore quite common edges. Note that QSYM has the biggest *Jaccard Distance* although it executes *Havoc* for 24 hours. The main reasons can be that 1) QSYM invests more computing resources, i.e., leveraging three threads running in parallel, and 2) QSYM leverages concolic execution [55] that may explore different paths compared with fuzzing. Furthermore, *MTFuzz* and *Neuzz* also have large *Jaccard Distance* mainly because they further use neural networks to guide the fuzzing process.

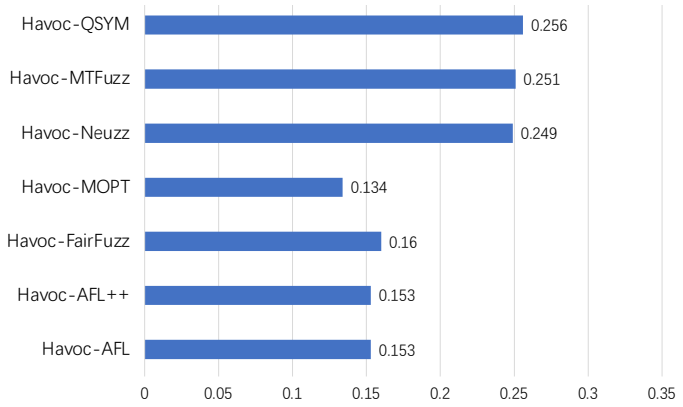


Figure 3.4: The average *Jaccard Distance* of different fuzzers in all studied programs.

> *Finding 7: Applying Havoc to different fuzzers potentially explores rather common edges, while fuzzers guided by concolic execution or neural networks can better complement Havoc.*

At last, we investigate the impact of appending *Havoc* on exposing program vulnerabilities. To this end, we attempt to collect the program crashes caused by executing the generated seeds with and without appending *Havoc* to all the studied fuzzers. Note that when we append *Havoc* to fuzzers, we ensure that it can be executed under sufficient time to fully leverage its power.

To begin with, it is essential to identify unique crashes since it is likely that many crashes are caused by the same program vulnerability. In this paper, we follow prior work [11, 94, 10, 21, 76, 171, 86] to identify the unique crashes only if they increase edge coverage. Note that in this paper, all of the crashes are explored by all of our previous evaluations. While a crash can only be reported once among all the fuzzing strategies (including *Havoc*) within a fuzzer, it can be possibly explored by different fuzzers. We then divide crashes into two sets, i.e., the ones explored by the involved *Havoc* mechanisms and the ones explored by the major fuzzing strategies. At last, we count the unique crashes for the two sets respectively.

Table 3.5 presents the results of the unique crashes. Overall, we derive 256 unique crashes from a total of 879 crashes where 243 (95%) are exposed by *Havoc* and 13 are

Table 3.5: The unique crashes found by *Havoc*

Programs	Crashes	Unique crashes	
		Havoc	Major
readelf (V2.30)	81	50	0
nm (V2.36)	89	78	1
objdump (V2.30)	1	1	0
size (V2.30)	4	2	1
strip (V2.30)	12	12	0
djpeg (V9c)	4	4	0
jhead (V3.04)	688	96	11
Total	879	243	13

exposed by their original fuzzing strategies, e.g., the constraint-solving-based mutations in QSYM and the gradient-driven mutations in *Neuzz*. Note that we exposed 69 unique cashes which have been fixed in the latest versions of their associated projects. We also report the rest unknown crashes (i.e., they can be exposed in the latest version) to the corresponding developers. The detailed bug report can be found in our GitHub page [124]. Moreover, applying *Havoc* can expose the crashes in 7 of the 12 total benchmark projects and be powerful in exposing unique crashes in projects `nm` (78 out of 79) and `jhead` (96 out of 107). Such facts indicate that applying *Havoc* can not only successfully advance program vulnerability exposure, but also potentially dominate the vulnerability exposure on certain projects.

Finding 8: Havoc can also play a vital role in exposing potential program vulnerabilities.

3.2 Enhancing *Havoc*

So far, our presented powerful performance of *Havoc* is simply caused by modifying its setups, including its execution time and integration modes with fuzzers. In this section, we attempt to investigate whether the power of *Havoc* can be further boosted. To this end, we first investigate the performance impact of the mutator stacking mechanism adopted by *Havoc*, and then propose an intuitive and lightweight technique to improve its performance accordingly.

3.2.1 Performance Impact of the Mutator Stacking Mechanism

Note that as a simplified mutation strategy, the mutator stacking mechanism contains two steps: determining *stacking size* and randomly selecting mutators, to impact the performance of *Havoc*. We then investigate the performance impact caused by each step. In particular, we first attempt to investigate the performance impact of *stacking size*. To this end, instead of randomly determining *stacking size* for mutating seeds at runtime of *Havoc* originally, we implement *Havoc* under a fixed *stacking size* for all its mutations. Figure 3.5 presents our evaluation results of the edge coverage ratio results in terms of all the possible fixed *stacking size*, i.e., 2, 4, 8,...128, on top of all the studied

benchmark projects. Note that the edge coverage ratio of one project is computed as the the explored edge number in terms of one fixed *stacking size* over the total explored edge number of all the fixed *stacking size*s. We can observe that overall, the *stacking size* which causes the optimal edge coverage performance for each studied project can be quite divergent, e.g., selecting *stacking size* 8, 2, and 32 can optimize the edge coverage in `tcpdump`, `djpeg`, and `mutool` respectively. Such results suggest that it is essential to adapt the *stacking size* setup for different projects to optimize their respective edge coverage.

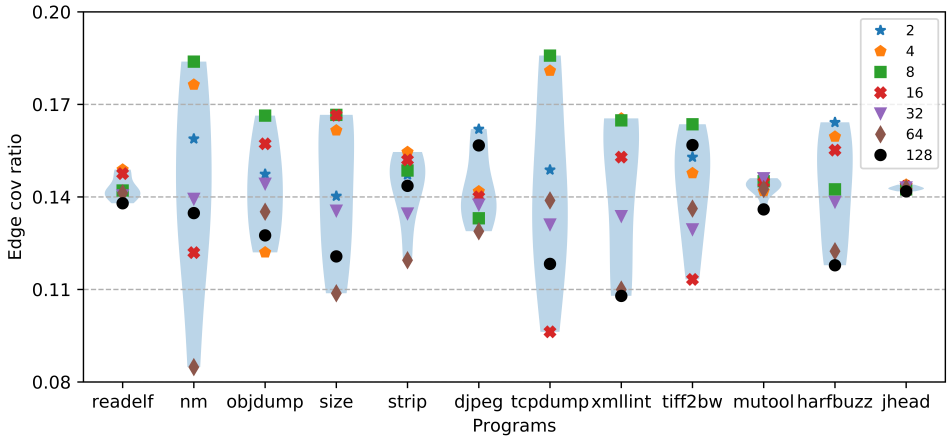


Figure 3.5: Edge coverage for different fixed stack sizes

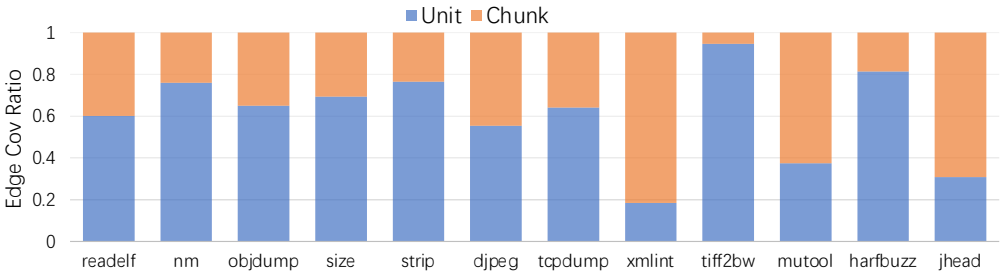


Figure 3.6: Edge coverage for *unit mutators* and *chunk mutators*

We then investigate the performance impact from mutators. To this end, instead of uniformly selecting mutators out of a total of 15 mutators, we first uniformly select *chunk mutators* or *unit mutators* and then randomly select their inclusive mutators under the given *stacking size* for mutating one seed. Figure 3.6 presents the edge coverage ratio results in terms of the selected mutator types on top of all the studied benchmark projects. Note that the edge coverage ratio is computed as the explored edge number by either *chunk mutators* or *unit mutators* over their total explored edge number. We can observe that overall, the distribution of the edge coverage ratio performance can be quite divergent among different projects, e.g., the edge coverage ratio of the *unit mutators* ranges from 18.39% (`xmllint`) to 94.53% (`tiff2bw`). Such results suggest that

it is also essential to adapt the selection of the mutator types for different projects to optimize their respective edge coverage performance.

3.2.2 Approach

Algorithm 1: The Framework of $Havoc_{MAB}$

```
Data: seed
Result: newseed
newseed ← seed;
stacksize ← selectStackArm();
mutatortype ← selectMutatorTypeArm(stacksize);
for iteration in stacksize do
    mutator ← randomSelectMutatorByType(mutatortype);
    newseed ← generateNewSeed(mutator, newseed);
reward ← 0;
if isInteresting(newseed) then
    reward ← 1;
updateStackBandit(reward, stacksize);
updateMutatorTypeBandit(reward, stacksize, mutatortype);
return newseed;
```

Inspired by the evaluation results above, we attempt to propose solutions to enhance *Havoc* via dynamically adjusting the project-wise selections on *stacking size* and mutators. Also, note that our previous findings reveal that to unleash the power of *Havoc*, it is essential to invest strong computing resources for *Havoc*. Accordingly, our design adopts the following principles. First, we only enable single thread/process, i.e., enhancing *Havoc* via only applying our specifically designed technique instead of leveraging more threads for more computing resource as found already. Second, our technique should be lightweight. In particular, when designing a technique for adjusting *Havoc* given the deterministic computing resource, ideally we aim for minimizing its overhead while maximizing the execution time for the *Havoc* mechanism itself.

In this chapter, we propose a lightweight single-threaded technique $Havoc_{MAB}$ (MAB refers to **M**ulti-**A**rmed **B**andit), for the pure *Havoc* to automatically adjust its selections on *stacking size* and mutators at runtime for facilitating its edge exploration. Specifically, we determine to model our task as a multi-armed bandit problem [155] which typically refers to allocating limited resources to alternative choices (i.e., *stacking size* and mutator selections for this problem) to maximize their expected gain (i.e., edge coverage for this problem). More specifically, we design a two-layer multi-armed bandit machine, i.e., a *stacking size*-level bandit machine and a mutator-level bandit machine, which is presented in Figure 3.7. Note that the *stacking size*-level bandit machine enables 7 arms where each arm is designed corresponding to a *stacking size* choice, i.e., 2, 4, 8,...128. After an arm of *stacking size* is chosen, the mutator-level bandit machine which enables 2 arms representing *chunk mutators* and *unit mutators* would first make a choice out of them and then proceed to select the exact mutators via uniform distribution. Eventually, $Havoc_{MAB}$ generates a mutant via the selected mutators and executes

it on the program under test for obtaining environmental feedback for further executions.

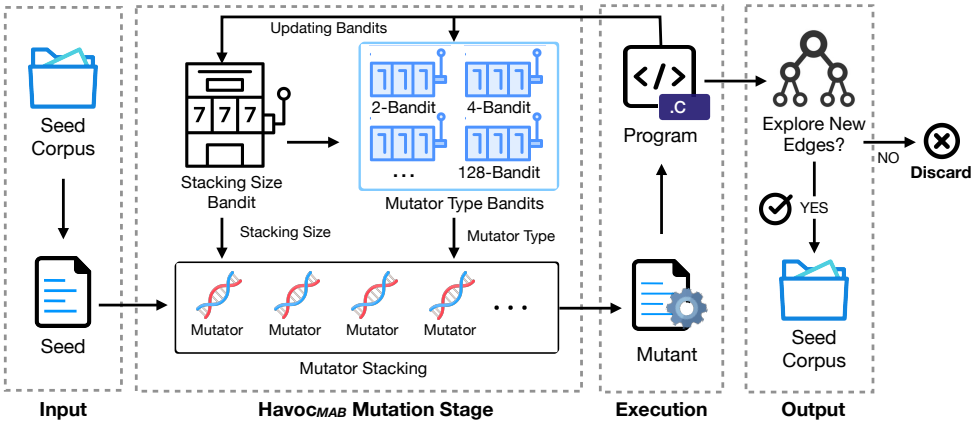


Figure 3.7: The Framework of $Havoc_{MAB}$

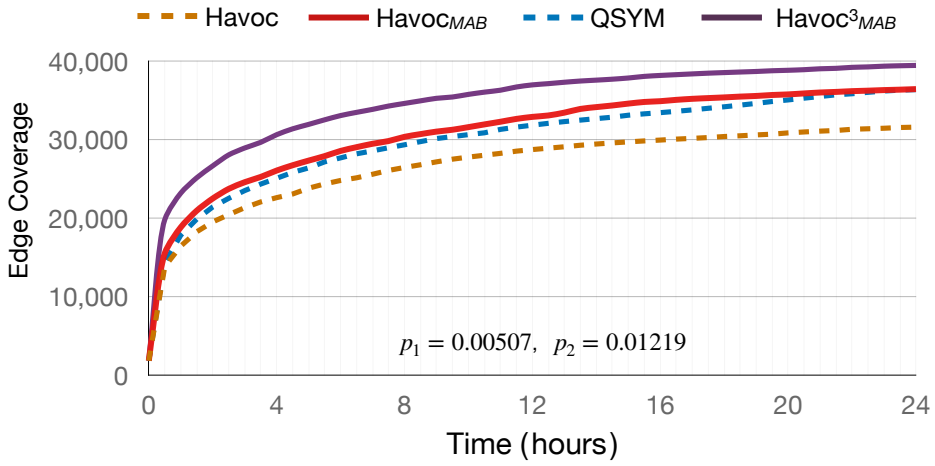


Figure 3.8: The average edge coverage of $Havoc_{MAB}$ over time

We adopt the widely-used UCB1-Tuned [7] algorithm to solve our proposed multi-armed bandit problem. Equation 3.1 demonstrates how to select an *arm* under such algorithm for a given bandit machine at time t. In particular, $\bar{x}_j$ refers to the average reward for arm_j till time t, n refers to the total execution count for the bandit machine and n_j refers to the execution count for arm_j, σ_j refers to the sample variance of arm_j.

$$arm(t) = \arg\max_j \left(\bar{x}_j + \sqrt{\frac{\ln n}{n_j} min(\frac{1}{4}, \sigma_j + \frac{2 \ln n}{n_j})} \right) \tag{3.1}$$

Note that we define the reward at time t as whether $Havoc_{MAB}$ has explored new edges or not for all the eight bandit machines. If a seed generated by a chosen *stacking size*

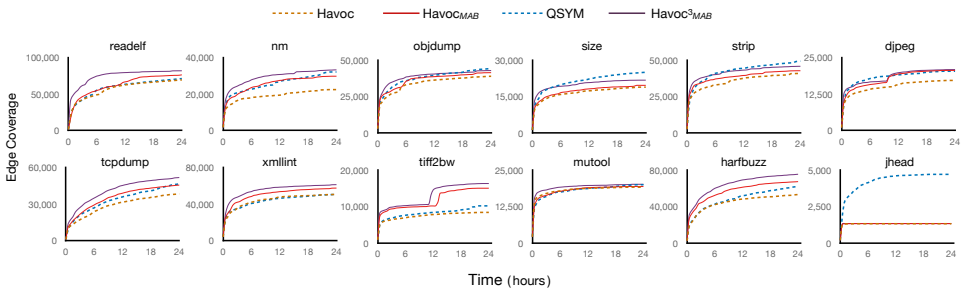


Figure 3.9: Edge coverage of $Havoc_{MAB}$ over time

and its selected mutators can explore new edges, the rewards returned to the *stacking size*-level bandit machine and its corresponding mutator-level bandit machine are both 1; otherwise, they are both 0.

Algorithm 1 presents our overall approach. $Havoc_{MAB}$ first selects *stacking size* for the executing seed and then selects its corresponding mutator type (Lines 2 to 3). Next, $Havoc_{MAB}$ generates a mutant by uniformly selecting the mutators of *stacking size* under the chosen type (Lines 4 to 6). Eventually, if such mutant explores new edges, we set the reward as 1 for its corresponding *stacking size*-level and mutator-level bandit machines (0 otherwise) to update Equation 3.1 for further executions (Lines 7 to 11).

3.2.3 Evaluation

To evaluate $Havoc_{MAB}$, we include QSYM for performance comparison since it presents the optimal edge coverage performance in our previous studies. Furthermore, we design a variant of $Havoc_{MAB}$ namely $Havoc^3_{MAB}$ where $Havoc_{MAB}$ is executed in three threads in parallel for comparing with QSYM under identical computing resources. We also include the pure *Havoc* as a baseline. We execute each variant for five times for each benchmark project to reduce the impact of randomness.

Figure 3.8 presents the average evaluation results of edge coverage of the studied approaches on top of all the benchmark projects under 24-hour execution. We can observe that $Havoc_{MAB}$ achieves significantly better performance than pure *Havoc*, i.e., increasing the average edge coverage among all the benchmark projects by 11.1% (34,574 vs 31,126 explored edges). Moreover, we apply the Mann-Whitney U test to illustrate the significance of $Havoc_{MAB}$. The fact that the p-value of $Havoc_{MAB}$ compared with *Havoc* in terms of the average edge coverage is 0.00507 indicates that $Havoc_{MAB}$ outperforms *Havoc* significantly ($p < 0.05$). Interestingly, although $Havoc_{MAB}$ only adopts one thread for execution, it can slightly outperform QSYM (which leverages three threads for execution) by 0.2% on average among all 5 runs with the STD of 108.55. It can also outperform QSYM for 4 out of 5 runs. On the other hand, executing $Havoc^3_{MAB}$ can result in 9% edge coverage gain over QSYM (37,614 vs 34,495 explored edges) with a p-value of 0.01219. Such results altogether can demonstrate the strength of our proposed $Havoc_{MAB}$.

Figure 3.9 presents the edge coverage trends of our studied approaches upon each benchmark for 24-hour execution. Overall, $Havoc_{MAB}$ outperforms pure *Havoc* in most of the benchmarks significantly. Moreover, $Havoc_{MAB}$ can outperform QSYM by at least 10% (60% more in `tiff2w`) in terms of edge coverage on five projects while incurring rather close performance on the rest projects with a single thread except `jhead`. Meanwhile, $Havoc^{3}_{MAB}$ can achieve the optimal edge coverage performance on eight benchmarks. Note that QSYM outperforms all other fuzzers in `jhead` (averagely 4,516 vs. 1,063). This demonstrates that grey-box fuzzing strategies alone are ineffective for `jhead` while the effectiveness can be largely improved by concolic execution leveraged in QSYM. Based on this observation and Finding 7, we highly recommend future research to investigate more powerful techniques for combining *Havoc*, concolic execution, and learning-based fuzzing.

3.3 Limitations

In this section, we summarize the limitations of this study's external and construct validity.

The threats to external validity mainly lie in the subjects and benchmarks. To reduce the threats, we select 8 representative state-of-the-art fuzzers, including AFL-based, concolic-execution-based, and neural program-smoothing-based fuzzers. We also adopt 12 benchmark projects according to their popularity, i.e., the most frequently used benchmarks by the original papers of our studied fuzzers. Another threat to external validity may lie in the randomness of the evaluation results. To reduce this threat, all the evaluation results are averaged upon five runs to reduce the impact of randomness.

The threat to construct validity mainly lies in the main metric used in this paper, i.e., edge coverage, to reflect code coverage. To reduce this threat, while there can be various ways to measure edge coverage, we choose to follow many existing fuzzers [86, 167, 130, 131, 21] and leverage the AFL built-in tool named `afl-showmap` for collecting edge coverage. Furthermore, we have also evaluated fuzzing effectiveness in terms of the number of unique crashes.

3.4 Summary

In this chapter, we investigate the impact and design of a random fuzzing strategy *Havoc*. We first conduct an extensive study to evaluate the impact of *Havoc* by applying *Havoc* to a set of studied fuzzers on real-world benchmarks. The evaluation results demonstrate that the pure *Havoc* can already achieve superior edge coverage and vulnerability detection compared with other fuzzers. Moreover, integrating *Havoc* to a fuzzer or extending total execution time for *Havoc* can also increase the edge coverage significantly. The performance gap among different fuzzers can also be considerably

reduced by appending *Havoc*. At last, we also design a lightweight approach to further boost *Havoc* by dynamically adjusting its mutation strategy.

Chapter 4

Fuzzing Study: the Neural Program-Smoothing-based Mechanism

To date, many existing approaches model fuzzing as an optimization problem and attempt to solve it by augmenting code coverage via mutating program seed inputs under a given time budget. Such *coverage-guided* fuzzing tasks can be typically resolved by applying search-based optimization algorithms such as evolutionary algorithms [165, 171, 37, 46, 143]. Specifically, test inputs are iteratively filtered, mutated, and executed such that the test results can approach the optimal solutions to satisfy the fitness functions of the adopted evolutionary algorithms, which are usually designed to maximize code coverage. However, evolutionary fuzzers have been argued that they fail to "leverage the structure (i.e., gradients or higher-order derivatives) of the underlying optimization problem" [131]. To address such issue, neural program-smoothing-based techniques, e.g., *Neuzz* [131] and *MTFuzz* [130], have been recently proposed to exploit the usage of gradients for fuzzing via neural network models. Specifically, they first adopt a neural network which, given the byte sequence of a seed as input, outputs a vector representing its associated program branching behaviors. Next, they compute the gradients of the collected output vectors with respect to the bytes of the given seed. Accordingly, they sort the resulting gradients and develop strategies to mutate the bytes with larger gradients more aggressively. Eventually, all the resulting mutants are used as test cases for fuzzing. Note that *MTFuzz* further attempts to outperform *Neuzz* by leveraging the power of multi-task learning and adopts a dynamic analysis module to augment the mutation strategy. In their original papers, *Neuzz* outperforms 10 existing coverage-guided fuzzers on 10 real-world projects by at least 3X more edge coverage over 24-hour runs and further detects 31 previously-unknown bugs. Compared to *Neuzz* and four other state-of-the-art fuzzers, *MTFuzz* achieves 2X to 3X edge coverage upon all the benchmark projects and exposes 11 previously-unknown bugs which cannot be detected by the other fuzzers.

Despite the effectiveness shown in their original papers, the evaluation on *Neuzz*

and *MTFuzz* can be potentially biased due to their limited benchmark suite with only 10 projects. Moreover, *Neuzz* and *MTFuzz* adopt a different edge coverage metric from many existing fuzzers [171, 11, 86, 94, 21] that can potentially bias the performance comparison. Furthermore, the investigation on the factors that can impact their edge coverage performance is rather limited, i.e., they only simply presented the overall effectiveness of the techniques without investigating the contributions made by individual components, e.g., the model structure, the gradient guidance mechanism, and the mutation strategy.

In this chapter, to enhance the understanding of the effectiveness and efficiency of program-smoothing-based fuzzing, we first construct a large-scale benchmark by retaining all the projects adopted in the original *Neuzz* and *MTFuzz* papers (except one that we fail to run) and adding 19 additional open-source projects that were frequently adopted in recent fuzzing research work. We then conduct an extensive evaluation for *Neuzz* and *MTFuzz* accordingly. The evaluation result suggests while *Neuzz* and *MTFuzz* can outperform AFL on all the studied benchmark projects by 10.5% and 8.9% on average in terms of edge coverage respectively, *MTFuzz* does not always outperform *Neuzz* and both their edge coverage performances are highly program-dependent. We also find neither their mutation strategies nor neural network models can be consistently effective. Meanwhile, although the gradient guidance mechanisms can be promising, their strengths have not been fully leveraged.

Inspired by the findings of our study, we propose an improved technique, namely *PreFuzz* [125], upon neural program-smoothing-based fuzzing. In particular, we develop a *resource-efficient edge selection mechanism* to facilitate the exploration on unexplored edges rather than the already covered edges. Moreover, we also apply a *probabilistic byte selection mechanism* guided by gradient information to *Neuzz* and *MTFuzz* to further boost edge exploration. Our evaluation results suggest that *PreFuzz* can significantly outperform *Neuzz* and *MTFuzz*, i.e., 43.1% more than *Neuzz* and 45.2% more than *MTFuzz* averagely in terms of edge coverage.

4.1 Extensive Study

4.1.1 Benchmarks

Although *Neuzz* and *MTFuzz* have been shown to outperform the existing fuzzers in terms of the edge coverage in the original papers [131, 130], such results can be possibly biased by the used subject projects. For example, 10 popular real-world projects are the main experimental subjects for both *Neuzz* and *MTFuzz*; however, it is not clear how such 10 projects are selected and whether the experimental findings can generalize to other real-world projects.

To reduce such threat, we extend the benchmark for evaluating *Neuzz* and *MTFuzz*. In particular, in addition to retaining the adopted 9 projects in the original papers (we could not successfully run project `Zlib` out of the 10 original projects), we also adopt

Table 4.1: Statistics of the studied benchmarks

Benchmark	Class	LOC	Package
bison	LEX & YACC	18,701	3.7
xmlwf	XML	6,871	expat-2.2.9
mupdf	PDF	123,562	1.12.0
pngimage	PNG	11,373	libpng-1.6.36
pngfix	PNG	12,173	libpng-1.6.36
pngtest	PNG	11,323	libpng-1.6.36
tcpdump	PCAP	46,892	4.99.0
nasm	ASM	18,941	nasm-2.15.05
tiff2pdf	TIFF	17,272	libtiff-4.2.0
tiff2ps	TIFF	16,177	libtiff-4.2.0
tiffdump	TIFF	15,113	libtiff-4.2.0
tiffinfo	TIFF	15,014	libtiff-4.2.0
libxml	XML	73,239	2.9.7
listaction	SWF	6,278	libming-0.4.8
listaction_d	SWF	6,272	libming-0.4.8
libsass	SCSS	14,638	libsass-3.6.5
jhead	JPEG	1,886	3.04
readelf	ELF	72,111	Binutils 2.30
nm	ELF	55,212	Binutils 2.30
strip	ELF	65,683	Binutils 2.30
size	ELF	54,463	Binutils 2.30
objdump	ELF	74,710	Binutils 2.30
libjpeg	JPEG	8,856	9c
harfbuzz	TTF	9,853	1.7.6
base64	FILE	40,332	LAVA-M
md5sum	FILE	40,350	LAVA-M
uniq	FILE	40,286	LAVA-M
who	FILE	45,257	LAVA-M

additional 19 projects for our extended evaluations. More specifically, to extend our benchmark projects, we first investigate all the fuzzing papers published in ICSE, ISSTA, FSE, ASE, S&P, CCS, USENIX Security, and NDSS in year 2020 and collect all their benchmark projects. Next, we sort the collected benchmark projects in terms of their usage in all the collected papers (presented in [125]). We then collect the top 30 most used benchmark projects and successfully run 19 of them which are eventually included in our extended benchmarks (the failed executions are mainly caused by environmental inconsistencies and unavailable dependencies). Table 4.1 presents the statistics of our adopted benchmarks. Specifically, we consider our benchmark to be sufficient and representative due to following reasons: (1) to the best of our knowledge, this is a rather large-scale benchmark suite compared with prior work; (2) the 28 collected benchmarks cover 12 different file formats for seed inputs, e.g., ELF, XML, and JPEG; and (3) the LoC of each program, ranging from 1,886 to over 120K, represents a wide range of program sizes.

4.1.2 Evaluation Setups

We conduct all our evaluations on Linux version 4.15.0-76-generic Ubuntu18.04 with RTX 2080ti. Following the evaluation setups of *Neuzz* and *MTFuzz*, for each selected

benchmark project, we first run AFL-2.57b on a single CPU core for 1 hour to initialize our seed collection and then run *Neuzz*, *MTFuzz* and all their variants (introduced in later sections) upon the collected seeds with a time budget of 24 hours. Note that all the edges within the 1-hour initial seed collection are excluded from the evaluation results in the remaining sessions. Moreover, we run our experiments for 5 times for each fuzzer and present the average results with close performance under different runs. Note that we instrument all the benchmark projects with `afl-gcc` to acquire runtime edge coverage.

In addition to studying *Neuzz* and *MTFuzz*, we also include AFL as a baseline technique throughout our extensive evaluations because (1) AFL is widely adopted as baseline by many fuzzing approaches [94, 11, 10, 168, 89] and frequently upgraded for improving its performance; and (2) *Neuzz* adopts multiple concepts originated from AFL for its implementation [129].

4.1.3 Research Questions

We investigate the following research questions to extensively study neural program-smoothing-based fuzzing.

- **RQ1:** How do *Neuzz* and *MTFuzz* perform on a large-scale dataset? For this RQ, we investigate their effectiveness and efficiency of edge exploration under our large-scale benchmark suite.
- **RQ2:** How do the key components of *Neuzz* and *MTFuzz* affect edge exploration? For this RQ, we attempt to investigate how exactly their adopted gradient guidance mechanisms, neural network models, and mutation strategies can affect edge exploration.

4.1.4 Results and Analysis

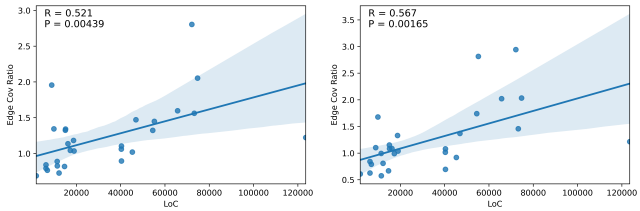
RQ1: performance of *Neuzz* and *MTFuzz* on a large-scale dataset.

We first investigate the edge coverage performance of all the studied fuzzers. In this chapter, following many existing coverage-guided fuzzers [171, 11, 86, 94, 21], we determine to adopt the number of the edges via `afl-showmap` as our default edge metric. Moreover, note that the edge metric of the original *Neuzz* and *MTFuzz* papers can be potentially biased since it counts the byte number of the `trace_bits` structure implemented by AFL and thus is inconsistent with the results provided by the guidance function (i.e., defining "interesting" seeds) in their implementations. Nevertheless, as a comprehensive study, we also evaluate all the studied fuzzers in terms of the edge metric of the original *Neuzz* and *MTFuzz* papers.

Table 4.2 presents the edge coverage results of our extensive study for *Neuzz* and *MTFuzz* under both adopted metrics. For instance, for AFL under `bison`, 10,374 corresponds to our default edge metric and 308 corresponds to the original metric in the

Table 4.2: Edge coverage results of all the studied approaches

Benchmarks	AFL	*Neuzz*	$Neuzz_{Rev}$	*MTFuzz*	$MTFuzz_{Rev}$	$MTFuzz_{Off}$	$Neuzz_{CNN}$	$Neuzz_{RNN}$	$Neuzz_{BRNN}$
bison	10,374 (308)	12,260 (432)	12,218 (264)	**13,812** (599)	12,799 (470)	12,801 (524)	12,375 (475)	12,592 (429)	12,444 (499)
xmlwf	**13,729** (3,272)	10,499 (2,200)	9,192 (1,644)	10,853 (509)	10,532 (457)	10,752 (476)	10,872 (2,794)	11,465 (2,891)	11,469 (2,831)
mupdf	13,665 (361)	16,705 (795)	**17,664** (654)	16,603 (328)	16,348 (237)	16,522 (334)	16,853 (796)	16,921 (792)	16,889 (804)
pngimage	**4,077** (201)	3,369 (324)	2,522 (199)	2,347 (200)	2,172 (198)	2,373 (194)	2,946 (241)	2,553 (197)	3,011 (323)
pngfix	**7,134** (135)	5,181 (85)	4,564 (71)	5,767 (80)	5,350 (71)	5,737 (73)	5,157 (74)	5,194 (74)	5,247 (76)
pngtest	3,185 (68)	2,828 (29)	2,933 (8)	3,166 (56)	2,719 (45)	3,016 (81)	3,074 (46)	**3,271** (58)	3,103 (42)
tcpdump	12,434 (3,525)	18,293 (4,310)	18,566 (5,005)	17,026 (5,512)	15,097 (3,715)	17,463 (4,621)	18,091 (4,495)	18,910 (4,635)	**19,411** (4,581)
nasm	33,633 (1,768)	34,788 (1,654)	33,838 (1,652)	34,958 (1,754)	34,451 (1,722)	33,907 (1,786)	**35,375** (1,791)	34,528 (1,695)	35,009 (1,899)
tiff2pdf	45,183 (4,844)	47,109 (4,365)	42,519 (3,993)	44,765 (4,355)	38,449 (3,676)	44,230 (4,044)	46,934 (3,971)	44,617 (3,765)	**50,347** (4,580)
tiff2ps	20,862 (3,621)	**23,705** (3,634)	21,063 (3,420)	22,671 (3,131)	16,700 (2,535)	21,817 (3,194)	23,931 (3,743)	21,322 (3,841)	23,160 (3,791)
tiffdump	2,416 (8)	**3,239** (52)	3,117 (52)	2,617 (64)	2,262 (38)	2,509 (61)	3,124 (46)	2,962 (47)	3,052 (43)
tiffinfo	11,964 (2,440)	**15,853** (2,618)	15,742 (2,407)	13,785 (2,799)	10,249 (1,431)	12,394 (1,904)	14,698 (2,788)	13,239 (2,649)	15,569 (2,379)
libxml	20,064 (541)	31,340 (1,553)	**32,075** (1,765)	29,236 (1,635)	29,162 (1,553)	27,902 (1,205)	31,421 (1,687)	31,774 (1,695)	31,731 (1,649)
listaction	**21,340** (3,151)	17,945 (2,893)	14,969 (2,328)	13,382 (622)	12,257 (559)	12,356 (591)	17,743 (2,791)	17,562 (2,822)	17,073 (2,770)
listaction_d	**31,617** (2,728)	25,006 (4,604)	18,643 (3,460)	26,629 (3,376)	21,644 (2,554)	23,619 (2,780)	25,869 (5,036)	28,436 (5,271)	23,622 (4,784)
libsass	**198,976** (10,385)	162,717 (8,492)	158,800 (8,438)	132,972 (7,373)	132,491 (7,232)	132,644 (7,106)	154,793 (8,742)	160,318 (8,902)	163,492 (8,527)
jhead	**2,082** (28)	1,433 (24)	1,566 (27)	1,268 (18)	1,215 (16)	1,273 (16)	1,327 (22)	1,560 (22)	1,502 (23)
readelf	14,329 (898)	40,186 (6,059)	34,994 (4,868)	42,173 (6,684)	35,178 (5,799)	40,005 (6,161)	42,889 (6,257)	**44,389** (6,159)	32,796 (5,314)
nm	11,154 (1,351)	16,159 (2,226)	13,505 (2,015)	**31,402** (3,707)	28,027 (3,128)	22,149 (3,045)	18,070 (2,312)	20,724 (3,132)	16,007 (4,099)
strip	20,536 (1,409)	32,791 (3,254)	31,604 (3,348)	**41,520** (5,172)	35,649 (5,699)	32,072 (3,674)	33,549 (3,649)	33,816 (3,916)	33,348 (3,753)
size	10,730 (1,188)	14,197 (2,450)	12,414 (2,036)	**18,675** (3,872)	16,525 (3,397)	12,623 (2,087)	14,488 (2,606)	14,254 (2,540)	12,284 (2,480)
objdump	15,492 (247)	31,808 (2,358)	28,617 (1,850)	31,507 (2,007)	27,227 (2,117)	30,074 (2,270)	31,176 (2,954)	33,165 (2,639)	**33,486** (3,062)
libjpeg	8,197 (266)	16,037 (1,600)	13,460 (1,566)	9,038 (876)	8,446 (797)	8,255 (487)	16,576 (1,729)	16,859 (1,713)	**17,566** (1,892)
harfbuzz	26,420 (3,107)	35,502 (5,982)	28,037 (5,606)	44,342 (6,268)	37,821 (4,930)	44,364 (6,155)	45,179 (7,542)	48,959 (7,656)	**50,911** (7,764)
base64	**1,344** (12)	1,202 (0)	987 (0)	935 (0)	912 (0)	819 (0)	1,247 (0)	1,226 (0)	1,243 (0)
md5sum	2,871 (33)	3,168 (131)	3,004 (37)	3,101 (35)	3,036 (35)	3,044 (35)	3,097 (33)	**3,241** (33)	3,163 (35)
uniq	713 (2)	**756** (2)	750 (2)	725 (0)	728 (0)	716 (0)	755 (2)	754 (2)	752 (2)
who	2,917 (14)	2,973 (17)	2,919 (17)	2,680 (15)	2,753 (17)	2,720 (17)	2,997 (17)	**3,031** (17)	3,026 (17)
Average	20,265 (1,640)	22,395 (2,219)	20,724 (2,026)	22,070 (2,180)	20,007 (1,872)	20,648 (1,890)	22,665 (2,380)	**23,130** (2,414)	22,883 (2,429)



(a) *Neuzz*/AFL **(b)** *MTFuzz*/AFL

Figure 4.1: Edge coverage advantage of the fuzzers over AFL

Neuzz/*MTFuzz* papers. For our default edge metric, we can observe that *Neuzz* significantly outperforms AFL by 10.5% (22,395 vs. 20,265 explored edges) in terms of edge coverage on average. Compared with the performance advantage claimed in its original paper (i.e., 2.7X), it is clearly degraded. We then investigate the performance difference among benchmark projects. Interestingly, we can observe that their performance advantage is rather inconsistent, i.e., ranging from -31.2% to 180.5%. Moreover, *Neuzz* only outperforms AFL upon 10 out of 19 extended projects. Such results suggest that *Neuzz* cannot always outperform AFL and the performance advantage of *Neuzz* over AFL can be program-dependent.

We also observe that *Neuzz* outperforms *MTFuzz* by 1.5% (22,395 vs. 22,070 explored edges) averagely in terms of edge coverage on all benchmark projects. While on 11 of 28 total projects, *MTFuzz* outperforms *Neuzz* by 20.8% averagely, *Neuzz* outperforms *MTFuzz* by 17.7% on the other 17 projects. Furthermore, even AFL outperforms *MTFuzz* by 33.6% averagely on a total of 11 projects. Such results indicate that similar to *Neuzz*, *MTFuzz* cannot perform consistently either.

We then attempt to reveal the characteristics of how the edge coverage performance varies among the studied projects. To this end, we delineate the correlation between the edge coverage advantage of the studied fuzzers compared with AFL and

the size of their studied projects via the Pearson Correlation Coefficient analysis. Figure 4.1 presents such results of *Neuzz* and *MTFuzz*. In each subfigure, the horizontal axis denotes the LoC of each project and the vertical axis denotes the ratio as dividing the edge coverage result of each studied approach by the edge coverage result of AFL. We can observe that overall, the correlation is rather strong (at the significance level of 0.05), i.e., all the studied approaches can result in larger edge coverage improvement over AFL upon larger projects than smaller ones. Such results clearly demonstrate that program size can significantly impact the edge coverage performance of neural program-smoothing-based fuzzers.

We observe similar data trends in terms of the edge metric in the original *Neuzz*/*MTFuzz* papers. In particular, *Neuzz* can outperform AFL by 35.3% (2,219 vs. 1,640 explored edges) and can outperform *MTFuzz* by 1.8% (2,219 vs. 2,180 explored edges). Note that under such measure, for certain projects, e.g., `base64`, *Neuzz* and *MTFuzz* explore zero edges after excluding the edges from 1-hour initial seed collection. Such results could be misleading that the studied fuzzers perform equally poor in `base64`, while such performance gaps can be clearly presented by our default edge metric.

> *Finding 1: The performance of Neuzz and MTFuzz can be largely program-dependent. Interestingly, such program-smoothing-based fuzzers tend to perform better on larger programs.*

Note that randomness is injected to many existing fuzzers [11, 94, 86, 168] for selecting bytes to guide mutations, e.g., AFL_{Havoc}. However, *Neuzz* and *MTFuzz* utilize only deterministic mutation strategies, i.e., adopting no randomness for selecting bytes which can be deterministically identified based on their corresponding gradient ranking. Therefore, we further investigate the edge exploration efficiency of random byte selection to infer whether including them in *Neuzz* and *MTFuzz* can be potentially beneficial. Specifically, we involve AFL in a fine-grained manner, i.e., its deterministic stage $AFL_{Deterministic}$ and the havoc stage AFL_{Havoc} (i.e., essentially the random byte selection mechanism) both of which enable non-deterministic execution time, for performance comparison with *Neuzz* and *MTFuzz*.

Figure 4.2 presents our evaluation results in terms of the explored edge number per second, namely *Edge Discovery Rate* (EDR) in this chapter, of *Neuzz*, *MTFuzz*, AFL, $AFL_{Deterministic}$ and AFL_{Havoc}. We can observe that overall, *Neuzz* and *MTFuzz* can outperform AFL by 10.2% and 8.5% respectively. Interestingly, AFL_{Havoc} achieves the highest EDR, i.e., 21.8X larger than $AFL_{Deterministic}$, 7.7X larger than *Neuzz*, and 7.8X larger than *MTFuzz* averagely on all the benchmarks. Accordingly, we can derive that AFL_{Havoc} can significantly augment edge exploration, i.e., it promptly explores edges upon the limited seed inputs provided by $AFL_{Deterministic}$. Such result is enlightening that applying random byte selection mechanism to neural program-smoothing fuzzers can potentially boost edge exploration.

Finding 2: AFL_{Havoc} dominates the efficiency of edge exploration, indicating that it is promising to augment edge exploration by adopting random byte selection mechanism.

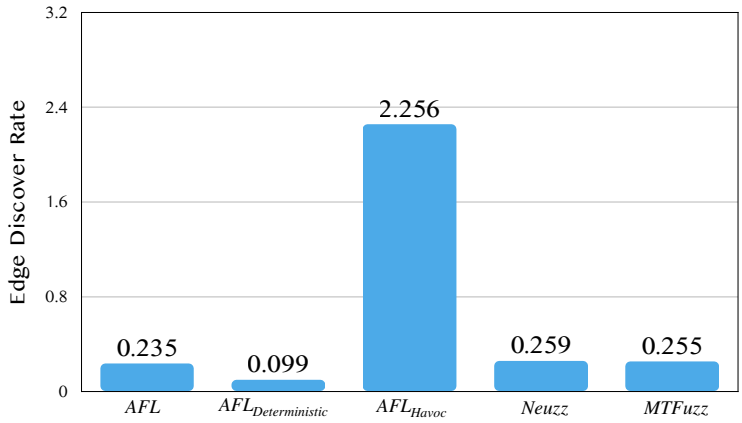


Figure 4.2: EDR of the studied approaches

RQ2: Effectiveness of the key components.

Gradient guidance. The inconsistencies between our finding and the declared results in the original papers (i.e., finding 1) inspire us to further investigate the performance impact of the adopted mechanisms of *Neuzz* and *MTFuzz*. To this end, we determine to first investigate the effectiveness of their dominating factor, i.e., the gradient guidance mechanism. In particular, since such mechanism is proposed to facilitate the mutations on the "promising" bytes for edge exploration via gradient computation, our purpose is to investigate whether their derived gradients can locate such bytes. More specifically, we propose an intuitive gradient guidance mechanism—instead of aggressively mutating the bytes with larger gradients in the original *Neuzz* and *MTFuzz*, we aggressively mutate the bytes with smaller gradients. Such mechanism is injected to *Neuzz* and *MTFuzz* to form their variants $Neuzz_{Rev}$ and $MTFuzz_{Rev}$. We thus evaluate $Neuzz_{Rev}$ and $MTFuzz_{Rev}$ to observe their performance difference from the original *Neuzz* and *MTFuzz* to investigate the effect of the gradient guidance mechanisms.

We can observe from Table 4.2 that *Neuzz* can explore 8.1% (1,671) more edges than $Neuzz_{Rev}$ and *MTFuzz* can explore 10.3% (2,063) more edges than $MTFuzz_{Rev}$ on average. Such consistent results suggest that larger gradients can be a better indicator to promising bytes, i.e., the derived gradients can reflect promising bytes.

Interestingly, $Neuzz_{Rev}$ can outperform *Neuzz* on 5 out of 28 projects, i.e., `libxml`, `mupdf`, `jhead`, `tcpdump` and `pngtest`. Meanwhile, $MTFuzz_{Rev}$ can outperform *MTFuzz* under `uniq` and `who`. Such results also indicate that the power of the gradient guidance in *Neuzz* and *MTFuzz* has not been completely leveraged.

Finding 3: Although the gradient guidance mechanisms adopted by Neuzz and MTFuzz are overall effective for identifying the promising bytes, their performance can be rather unstable on some programs.

DNN models. Now that the gradients derived by *Neuzz* and *MTFuzz* can be proven to be effective in reflecting promising bytes for mutations, we further investigate how their corresponding neural network models impact edge exploration. Specifically, since compared to *Neuzz*, *MTFuzz* enables the independent dynamic analysis module *Crack* to augment their mutation strategy, we turn it off and form its variant $MTFuzz_{Off}$, i.e., applying the mutation strategy of *Neuzz* in *MTFuzz*, such that they only differ in the adopted neural network models. Moreover, we also include the Convolutional Neural Network (CNN) [84] model and two commonly-used Recursive Neural Network (RNN) [45] models, i.e., LSTM [61] and Bi-LSTM [57], and adopt them in the original *Neuzz* to form its variants $Neuzz_{CNN}$, $Neuzz_{RNN}$, and $Neuzz_{BRNN}$. Note that we investigate more RNN-based models since they are typically used in learning the distribution over a sequence to predict the future symbol sequence [29] (e.g., for speech recognition) and expected to better match the program input features than CNN-based models. Eventually, we determine to evaluate *Neuzz* and all the variant techniques to detect how multiple neural network models impact the edge exploration of program-smoothing-based fuzzers. Note that their hyper-parameter setups are introduced in our GitHub page [125].

We can observe from Table 4.2 that overall, all our studied approaches perform similarly in terms of edge coverage. Specifically, *Neuzz* slightly outperforms $MTFuzz_{Off}$ by 8.5% (22,395 vs 20,648 explored edges), underperforms $Neuzz_{CNN}$ by 1.2% (22,395 vs. 22,665 explored edges), $Neuzz_{RNN}$ by 3.3% (22,395 vs. 23,130 explored edges) and $Neuzz_{BRNN}$ by 2.2% (22,395 vs. 22,883 explored edges). Meanwhile, we can also observe that none of the studied approaches can dominate on top of all the studied projects, i.e., *Neuzz* dominates 7, $MTFuzz_{Off}$ dominates 2, $Neuzz_{CNN}$ dominates 4, $Neuzz_{RNN}$ dominates 9, and $Neuzz_{BRNN}$ dominates 6. Therefore, we derive that upgrading neural network models cannot significantly impact the performance of edge exploration.

Finding 4: Different neural network models have limited impact on the effectiveness of program-smoothing-based fuzzing.

Mutation Strategies. We then investigate the impact from the mutation strategy of the neural program-smoothing-based fuzzers. Specifically, since *MTFuzz* differs from *Neuzz* mainly by enabling *Crack* for mutations and their respective neural network models do not significantly impact the edge exploration (reflected by Finding 4), we concentrate our investigation on the impact from *Crack*. To this end, we evaluate *MTFuzz* and $MTFuzz_{Off}$. Table 4.2 demonstrates that overall, *MTFuzz* can outperform $MTFuzz_{Off}$ by 6.9% (22,070 vs. 20,648 explored edges). However, such advantage can be rather inconsistent, ranging from -2.5% to 47.9% upon individual projects. On the

other hand, applying *Crack* can be potentially cost-ineffective since it is quite heavy-weight. Therefore, it is essential to consider whether it is worthwhile in applying such technique for neural program-smoothing-based fuzzing.

> *Finding 5: The dynamic analysis module Crack adopted by MTFuzz can be cost-ineffective.*

4.1.5 Discussion

We first discuss why neural network models do not significantly impact the edge coverage performance. To this end, we ought to understand the effect of the adopted neural network models of *Neuzz* and *MTFuzz*. In particular, note that neural networks are usually used for data prediction, i.e., learning and generalizing historical data to predict unseen data. Accordingly, researchers have developed many neural network models to strengthen their generalization and prediction capabilities. Therefore, one may misunderstand that *Neuzz* and *MTFuzz* attempt to use neural network models to predict the bytes corresponding to unexplored edges. Instead, as a matter of fact, *Neuzz* and *MTFuzz* leverage neural network models which compute the gradients to reflect the relations between explored edges and seed inputs, i.e., mutating the byte corresponding to a larger gradient can be more likely to explore a new edge other than the existing edge under one shared prefix edge. As a result, any neural network model can be applied as long as it can successfully deliver gradients to reflect such *explored edge—seed input* relations, i.e., how its generalization or prediction capability does not quite matter under such scenarios. Therefore, it is quite likely that a simplistic model (e.g., feed-forwarded network model adopted by *Neuzz*) can perform similarly as fine-grained models (e.g., multi-task learning model adopted by *MTFuzz* and the RNN models adopted by the studied *Neuzz* variants).

We then attempt to illustrate why *Neuzz* and *MTFuzz* cannot always be effective. Note that even though *Neuzz* and *MTFuzz* enable gradient guidance mechanisms to explore new edges, their iterative training-and-mutation strategy via randomly selecting edges and seeds in the beginning can nevertheless select existing edges other than unexplored edges to compute gradients, i.e., they still allow inefficient mutations. Specifically for the smaller programs where *Neuzz* and *MTFuzz* cannot outperform AFL, their edge exploration converges faster than larger programs due to the limited number of edges, i.e., they have a higher chance to select an existing edge whose "sibling" edges have already been explored by other seeds for gradient computation. Thus, it can be difficult to mutate its "promising" bytes for exploring new edges.

4.2 *PreFuzz*

Our findings reveal that we can possibly leverage the power of the gradient guidance mechanism to enhance the edge exploration of neural program-smoothing-based

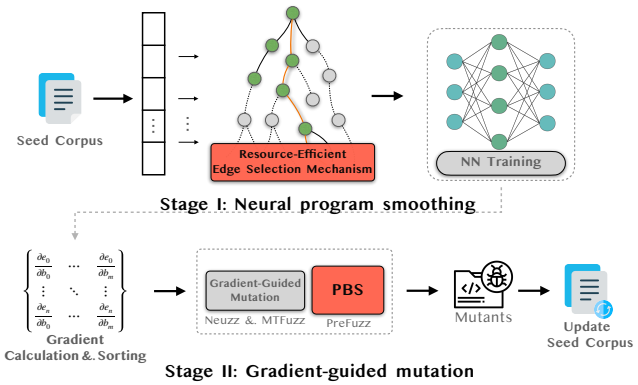


Figure 4.3: Framework of *PreFuzz*

fuzzers. To this end, we propose *PreFuzz* (**P**robabilistic **re**source-efficient program-smoothing-based **Fuzz**ing). Figure 4.3 presents the workflow of *PreFuzz*. *PreFuzz* first trains a neural network model by applying all the existing seeds as the training set. Next, *PreFuzz* adopts a *resource-efficient edge selection mechanism* to select edges for gradient computation. Then, the gradient information is utilized to generate mutants for fuzzing. Note that a mutant which explores new edges can be used as a seed for further edge exploration. Meanwhile, *PreFuzz* adopts *probabilistic byte selection mechanism* (*PBS* in Figure 4.3) to facilitate mutations.

4.2.1 The Details

Resource-Efficient Edge Selection Mechanism

The purpose of the *resource-efficient edge selection mechanism* is to prevent exploring the existing branching behaviors (i.e., edges). To this end, our mechanism is designed to identify the edge worthy being explored for later selecting and mutating its corresponding byte. Intuitively, when one edge can identify the number of its "sibling" edges, such edge number can be a potential indicator whether the given edge should be included for further gradient computation. More specifically, the more "sibling" edges have been explored, the less likely new "sibling" edges can be explored via the gradient computation for the given edge.

Algorithm 2 presents the details of the *resource-efficient edge selection mechanism*. First, it is quite essential to acquire the runtime edge exploration states, e.g., the number of "sibling" edges of a given edge and how many have been explored (lines 1 to 2). To this end, we decompile the assembly-level programs, parse them to the instructions via AFL-specific instrumentation, and construct the edge exploration states via statically analyzing the parsed instructions. Next, given one edge, we derive the ratio of its explored "sibling" edge number over its total "sibling" number (lines 4 to 8). If such ratio is lower than a preset threshold, we retain the given edge and stores it in a *Candidate Edge Set* where we later randomly select such edges for further gradient

Algorithm 2: Candidate Edge Set Construction

```
   Data: threshold, exploredEdge
   Result: selectedEdges
1  candidate ← set();
2  correspRelation ← getEdgeRelation();
3  for edge in exploredEdge do
4  |   explored ← 0;
5  |   siblings ← |correspRelation[edge]|;
6  |   for neighbour in correspRelation[edge] do
7  |   |   if neighbour in exploredEdge then
8  |   |   |   explored ← explored + 1;
9  |   if explored/siblings < threshold then
10 |   |   candidate.add(edge);
11 selectedEdges ← randomlySelectFromSet(candidate);
12 return selectedEdges;
```

computation (lines 9 to 11). We use Figure 2.3 to further illustrate such algorithm. Assuming that e_0 can be explored given the "seed" in Figure 2.3, mutating the byte of the given seed corresponding to the access condition of e_0 can explore its "sibling" edge e_1. While *Neuzz* and *MTFuzz* are designed to perform such mutation for edge exploration, e_1 could have nevertheless been explored already due to the randomness injected to their mechanisms. Thus, the effectiveness of the gradient guidance mechanism may be compromised. However, our *resource-efficient edge selection mechanism* can collect the exploration information of the "sibling" edge of e_0, i.e., e_1, before computing the gradient for e_0. If it finds out that e_1 has already been explored, it would not select e_0 for gradient computation in the first place to save the computing resource.

Probabilistic Byte Selection Mechanism

Inspired by Finding 2, we further inject an additional non-deterministic stage to neural program-smoothing fuzzers. To this end, we develop a *Probabilistic Byte Selection Mechanism* and append it to *Neuzz* to expand edge exploration. Note that the *probabilistic byte selection mechanism* utilizes the gradient information generated by the *resource-efficient edge selection mechanism*, and gets activated after the mutation stage inherited from *Neuzz*. This stage contains three steps: (1) dividing each seed input into segments, (2) selecting segments by gradient-based probability distribution, and (3) randomly selecting bytes from the selected segment for mutation via AFL_{Havoc} mutators.

Unlike AFL_{Havoc} which randomly selects bytes from the whole seed, we first divide a seed into a constant number (8 by default in our paper) of equal-length segments. We then select seed segments based on their probabilities. Note that while intuitively leveraging byte-wise probability distribution for byte selection is more natural, this is essentially deterministic and excludes the benefits of randomness (as in Finding 2). Therefore, our probability distribution is established upon seed segments rather than individual bytes so as to leverage the power of randomness and AFL_{Havoc}.

Table 4.3: Edge coverage results of *PreFuzz*

Benchmarks	AFL	*Neuzz*	*MTFuzz*	$Neuzz_{EdgeSelection}$	$Neuzz_{Prob}$	*PreFuzz*
bison	10,374	12,260	13,812	13,003	13,744	**15,078**
xmlwf	13,729	10,499	10,853	12,290	16,960	**21,009**
mupdf	13,665	16,705	16,603	17,002	19,995	**21,203**
pngimage	4,077	3,369	2,347	2,883	2,838	**4,876**
pngfix	7,134	5,181	5,767	7,307	7,422	**7,930**
pngtest	3,185	2,828	3,166	3,993	4,199	**4,703**
tcpdump	12,434	18,293	17,026	19,764	30,767	**34,947**
nasm	33,633	34,788	34,958	37,534	41,973	**43,628**
tiff2pdf	45,183	47,109	44,765	51,506	50,555	**57,172**
tiff2ps	20,862	23,705	22,671	23,649	24,247	**29,332**
tiffdump	2,416	3,239	2,617	3,590	3,554	**3,888**
tiffinfo	11,964	15,853	13,785	17,157	17,572	**21,839**
libxml	20,064	31,340	29,236	32,161	40,935	**47,689**
listaction	21,340	17,945	13,382	20,208	29,161	**32,447**
listaction_d	31,617	25,006	26,629	26,351	40,355	**46,762**
libsass	198,976	162,717	132,972	169,936	215,510	**218,130**
jhead	2,082	1,433	1,268	1,952	2,463	**2,464**
readelf	14,329	40,186	42,173	40,396	47,727	**53,859**
nm	11,154	16,159	**31,402**	19,040	22,605	30,709
strip	20,536	32,791	41,520	33,864	37,705	**42,943**
size	10,730	14,197	18,675	14,734	15,261	**19,231**
objdump	15,492	31,808	31,507	34,036	38,983	**43,195**
libjpeg	8,197	16,037	9,038	17,192	22,615	**24,365**
harfbuzz	26,420	35,502	44,342	47,877	45,412	**60,333**
base64	1,344	1,202	935	1,454	1,631	**1,644**
md5sum	2,871	3,168	3,101	3,415	**3,580**	3,518
uniq	713	756	725	792	794	**795**
who	2,917	2,973	2,680	3,262	3,255	**3,491**
Average	20,265	22,395	22,070	24,155	28,636	**32,042**

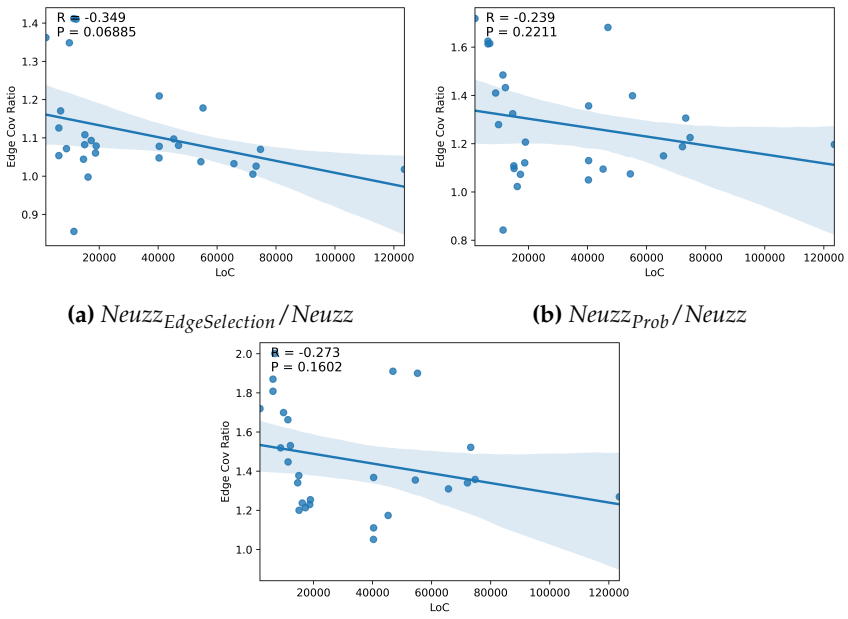
Next, we calculate the fitness score for each segment, presented in Equation 4.1, where $\sum_{j=1}^{seg_i} grad_j$ denotes the gradient sum for all the bytes within a given segment seg_i, $length(seg_i)$ denotes its byte number, and the fitness score for a given segment seg_i is computed as the average gradient of all the bytes within seg_i.

$$fitness_{seg_i} = \frac{\sum_{j=1}^{seg_i} grad_j}{length(seg_i)} \tag{4.1}$$

Accordingly, the probability $Prob_{seg_i}$ for selecting a segment seg_i for mutation is presented in Equation 4.2, i.e., the ratio of the fitness score of seg_i over the total fitness scores of all the segments.

$$Prob_{seg_i} = \frac{fitness_{seg_i}}{\sum_{j=1}^{total} fitness_{seg_j}} \tag{4.2}$$

Finally, we apply AFL_{Havoc} to mutate the selected segments. In particular, AFL_{Havoc} randomly selects a byte from the segment for mutation based on its mechanism. Note that if the mutants are also "interesting", they are retained for further gradient computation and the *probabilistic byte selection mechanism*. Such process is iterated until hitting the time budget.



(a) $Neuzz_{EdgeSelection}$/*Neuzz* **(b)** $Neuzz_{Prob}$/*Neuzz*

(c) *PreFuzz*/*Neuzz*

Figure 4.4: Edge coverage ratio upon *Neuzz*

4.2.2 Performance Evaluation

We attempt to evaluate the performance of *PreFuzz* and its technical components respectively. To evaluate the usage of the *resource-efficient edge selection mechanism* and the *probabilistic byte selection mechanism*, we form two *Neuzz* variants, i.e., $Neuzz_{EdgeSelection}$ which injects *resource-efficient edge selection mechanism* to *Neuzz* and $Neuzz_{Prob}$ which appends the *probabilistic byte selection mechanism* to *Neuzz*. Note that we retain *Neuzz*, *MTFuzz*, and AFL as our baselines for performance comparison. The experimental setups in this section follow the same settings in Section 4.1.2. The `threshold` for Algorithm 2 is set to 0.4[1].

Edge exploration effectiveness.

Table 4.3 presents the experimental results of edge exploration effectiveness. We can find that overall, *PreFuzz* outperforms all the existing baselines in terms of edge coverage averagely, e.g., *PreFuzz* can outperform AFL by 58.1% (32,042 vs. 20,265 explored edges) and *Neuzz* by 43.1% (32,042 vs. 22,395 explored edges). Note that under the originally adopted metric of edge coverage, *PreFuzz* also outperforms *Neuzz* and *MTFuzz* by 34.3% and 36.7%. Such results suggest that combining the *resource-efficient edge selection mechanism* and the *probabilistic byte selection mechanism* for *Neuzz* can be rather powerful. Moreover, $Neuzz_{EdgeSelection}$ outperforms *Neuzz* by 7.9% (24,155 vs. 22,395 explored edges) and *MTFuzz* by 9.4% (24,155 vs. 22,070 explored edges). Specifically, *Neuzz* obtains 271 more edges averagely than $Neuzz_{EdgeSelection}$ on 2 projects while

[1]We also evaluate more threshold setups and present the results in our GitHub link [125] which indicates that changing threshold setups incurs limited performance impact.

$Neuzz_{EdgeSelection}$ obtains 1,917 more edges averagely than *Neuzz* on the rest 26 projects. Such results indicate that the *resource-efficient edge selection mechanism* can enhance the overall effectiveness of *Neuzz*. In addition, $Neuzz_{Prob}$ also outperforms *Neuzz* by 27.9% (28,636 vs. 22,395 explored edges) and *MTFuzz* by 29.8% (28,636 vs. 22,070 explored edges). Such results demonstrate that introducing randomness can also significantly increase the edge coverage of the neural program-smoothing-based fuzzers.

Figure 4.4 presents the correlation between the edge coverage advantage of *PreFuzz*, $Neuzz_{Prob}$, $Neuzz_{EdgeSelection}$ over *Neuzz* by dividing their corresponding edge coverage results and the LoC of the studied benchmark projects. Interestingly, we can observe that the correlation is rather weak, i.e., all presented p values (0.0688, 0.2211 and 0.1602) fail to reach the significance level of 0.05. It indicates that the edge coverage advantage over the original *Neuzz* is not affected by the program size. Moreover, such advantage is rather consistent. Specifically, we determine to use Coefficient of Variation (CV) [14], a widely-used metric for measuring the dispersion of a probability distribution [164, 119, 123], to measure the consistency of their performance improvement. Note that a lower CV indicates a more consistent performance improvement. As a result, *PreFuzz*, $Neuzz_{EdgeSelection}$, and $Neuzz_{Prob}$ can achieve 19.6%, 11.5%, and 17.4% of CV for their performance improvement over *Neuzz*, which are all significantly reduced compared with the CV of *Neuzz* (37.6%) for its improvement over AFL. Therefore, we summarize that our proposed mechanisms can significantly and consistently strengthen the neural program-smoothing-based fuzzers. Note that we find under the edge coverage metric adopted in the original *Neuzz*/*MTFuzz* papers, the performance gain of *PreFuzz* over *Neuzz* is 34.3% (2,981 vs. 2,219 explored edges) which is also rather significant.

Figure 4.5 presents the average time trend of edge coverage within 24 hours for AFL, *MTFuzz*, *Neuzz* and *PreFuzz* among all the benchmark projects. We can observe that at any time being, *PreFuzz* can outperform other fuzzers significantly in terms of edge coverage.

In-depth Ablation Study

In this section, we further perform in-depth ablation studies to investigate the efficacy of our *resource-efficient edge selection mechanism* and *probabilistic byte selection mechanism* respectively. Specifically for the *resource-efficient edge selection mechanism*, we find that overall, 24.0% edges do not need to be explored by applying $Neuzz_{EdgeSelection}$ under each iteration averagely (1,230 vs. 935 edges). Moreover, the *probabilistic byte selection mechanism* in *PreFuzz* is more efficient when combining with the *resource-efficient edge selection mechanism* since *PreFuzz* explores averagely 11.9% more edges than $Neuzz_{Prob}$ (32,042 vs. 28,636 explored edges in Table 4.3). Such results indicate that applying the *resource-efficient edge selection mechanism* can significantly save the effort on exploring the edges which cannot contribute to increasing edge coverage.

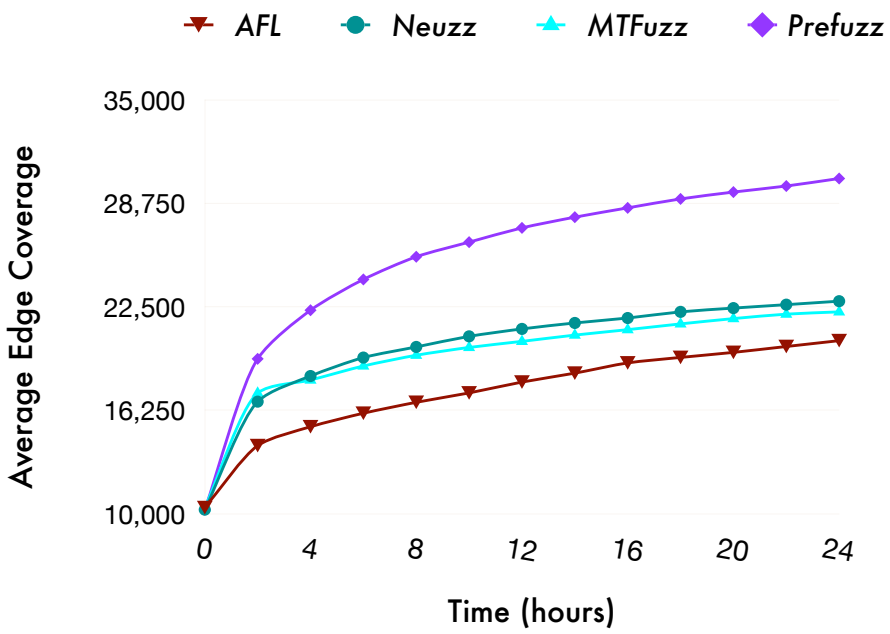


Figure 4.5: Edge coverage of *PreFuzz* in terms of time

We further investigate the *probabilistic byte selection mechanism* in terms of *Edge Discovery Rate*. To this end, we also include AFL_{Havoc}, $Neuzz_{EdgeSelection}$, the gradient-guided mutation stage of *PreFuzz*, and the probabilistic byte selection stage of *PreFuzz* (represented as $PreFuzz_{Gradient}$ and $PreFuzz_{Prob}$, respectively) for performance comparison. Note that $PreFuzz_{Gradient}$ and $PreFuzz_{Prob}$ results are extracted from the two stages of a complete *PreFuzz* run, e.g., $PreFuzz_{Prob}$ utilizes the *resource-efficient edge selection mechanism* to select edges for computing their gradients while $Neuzz_{Prob}$ randomly selects explored edges for gradient computation. Figure 4.6 presents our evaluation results. We can observe that overall, $PreFuzz_{Prob}$ can significantly outperform all the other studied approaches on top of all the studied benchmarks, e.g., $PreFuzz_{Prob}$ can be 62.0% more efficient than AFL_{Havoc} (3.656 vs. 2.256 EDR). Accordingly, we can infer that the gradient guidance adopted by *PreFuzz* can provide more "high-quality" seeds and more efficient guidance (i.e., gradients) for launching its *probabilistic byte selection mechanism* to explore more new edges than AFL_{Havoc}. Furthermore, we can observe that the EDR of $PreFuzz_{Gradient}$ can also outperform the original $Neuzz_{EdgeSelection}$ by 91.4%. Therefore, we also infer that $PreFuzz_{Prob}$ can advance the edge exploration efficiency of $PreFuzz_{Gradient}$. To summarize, combining the two improvements can mutually advance their edge exploration.

Crashes

Table 4.4 presents the unique crashes exposed by *Neuzz*, *MTFuzz* and *PreFuzz* in the studied benchmarks. Overall, *PreFuzz* explores the most unique crashes by outperforming *Neuzz* by 62% (149 vs. 92), and *MTFuzz* by 80% (149 vs. 83). In addition,

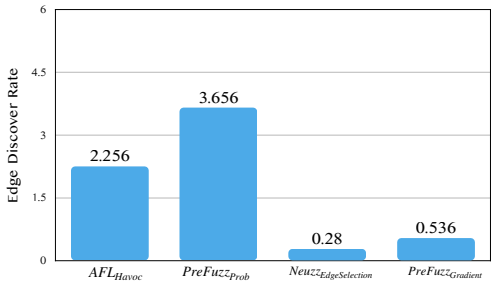


Figure 4.6: *Edge Discovery Rate* of different *PreFuzz* stages

Table 4.4: Unique crashes found by *Neuzz*, *MTFuzz* and *PreFuzz*

Benchmarks	*Neuzz*	*MTFuzz*	*PreFuzz*
size	5	9	7
readelf	15	7	37
libjpeg	2	0	5
objdump	0	0	1
who	2	0	4
bison	15	18	20
jhead	8	7	12
listaction	25	16	31
listaction_d	7	8	17
nm	3	3	3
strip	10	15	12
Total	92	83	149

PreFuzz dominates the number of the exposed unique crashes in each benchmark. Furthermore, the crashes exposed by *Neuzz* and *MTFuzz* are also detected by *PreFuzz* in our evaluation. Such results suggest that *PreFuzz* can also be more effective than *Neuzz* and *MTFuzz* in terms of exposing potential vulnerabilities.

4.2.3 Implications

Based on our findings in this chapter, we propose the following implications for advancing the future research on fuzzing.

Simplistic neural network models may suffice. Our study results reveal that the edge coverage performance can be essentially impacted by how the resulting gradients of the adopted neural network models reflect the relations between explored edges and seed inputs rather than their generalization or prediction capabilities. That said, simplistic neural network models may already suffice for program-smoothing-based fuzzing.

Think twice before applying dynamic analysis. Our evaluations indicate that the dynamic analysis module adopted in *MTFuzz* can be quite effective on large programs. However, executing such a module can be rather heavyweight, similar to many other program analysis techniques[35, 15, 56]. Therefore, we recommend thinking carefully

before adopting dynamic analysis techniques to enhance neural program-smoothing-based fuzzing.

Edge selection? Yes! Gradient computation? Maybe. Our evaluations reveal that selecting "promising" edges for mutations can be quite effective in increasing the edge coverage performance on programs of varying sizes. Meanwhile, one question can be asked: is it necessary to bind such powerful mechanism with gradient guidance? Especially when we realize that the power of neural networks can be argued to be "underused" (i.e., their generalization and prediction capabilities are underused), such question can then be transformed as — is it necessary to use neural networks for computing gradients to represent the relations between explored edges and seed inputs? To answer such question, it is worthwhile to attempt other lightweight alternatives to represent such relations as potential future research directions.

Probabilistic search helps. Our study results indicate that the edge coverage performance of the neural program-smoothing-based fuzzers can be significantly enhanced by appending the *probabilistic byte selection mechanism*. Intuitively, we suggest the users to design such probabilistic search strategy with more guidance to any of their adopted fuzzers when possible. Accordingly, one possible research direction can be how to integrate such probabilistic search strategy with diverse fuzzers for optimizing the edge coverage performance.

4.3 Limitations

In this section, we summarize the limitations of this study's external and construct validity.

The threat to external validity mainly lies in the benchmarks used. To reduce this threat, we adopt the original benchmarks used by *Neuzz* and *MTFuzz*, and add 19 more projects widely used for the evaluations in many popular fuzzers [94, 11, 10, 167, 89] published recently.

The threat to construct validity mainly lies in the metrics used. While the edge coverage metrics adopted by *Neuzz* and *MTFuzz* are not widely used by the existing fuzzers and can be arguably limited to reflect edge coverage, to reduce this threat, we determine to follow the majority by using the AFL built-in tool *afl-showmap* for measuring edge coverage while also presenting partial results in the original measure as well. Notably while under our metric, the performance advantages of *Neuzz* and *MTFuzz* are reduced, our *PreFuzz* can incur quite strong performance gain under both metrics.

4.4 Summary

In this chapter, we investigate the strengths and limitations of neural program-smoothing-based fuzzing approaches, e.g., *MTFuzz* and *Neuzz*. We first extend our benchmark suite by including additional projects that were widely adopted in the existing fuzzing

evaluations. Next, we evaluate *Neuzz* and *MTFuzz* on the extensive benchmark suite to study their effectiveness and efficiency. Inspired by our study findings, we propose *PreFuzz* combining two technical improvements, i.e., the *resource-efficient edge selection mechanism* and the *probabilistic byte selection mechanism*. The evaluation results demonstrate that *PreFuzz* can significantly outperform *Neuzz* and *MTFuzz* in terms of edge coverage. Furthermore, our results also reveal various findings/guidelines for advancing future fuzzing research.

Chapter 5

Enhancing Coverage-Guided Fuzzing via Phantom Program

Albeit many coverage-guided fuzzers have been shown effective in terms of code coverage and bug exposure [159, 21, 90, 77], their coverage-guided strategies are still somewhat restricted to hinder their further performance improvement. In particular, the existing coverage-guided fuzzing strategies typically require complete execution on each seed, i.e., exploring program states bounded by rigorous dependencies between program branches (referred to as *program dependencies* in the rest of the chapter for simplicity). It has been widely shown that in this way, many seeds are executed to only result in the limited state exploration of target programs [86, 132, 172, 173], indicating that a large number of such seeds are ineffective in exposing new program states. Furthermore, even for the limited number of seeds which can effectively explore program states, their iterative executions are still subject to rigorous program dependencies, i.e., incrementally exploring program states in order. As a result, the fuzzers have to repeatedly access the covered program states before uncovering new program states under each iterative execution. Such facts indicate that the seed-wise exploration power on program states has not been sufficiently leveraged by the existing coverage-guided fuzzing strategies.

In this chapter, we attempt to tackle the aforementioned limitations of the seed-wise exploration power for the existing coverage-guided fuzzing strategies. Our key insight is that instead of only using a limited number of effective seeds for incrementally exploring program states under iterative executions, we seek to exploit more effective seeds as well as the exploration on separate program states by reducing their inter-dependencies so as to enhance the efficacy of coverage-guided fuzzing. Accordingly, we propose *MirageFuzz*, the first fuzzer which attempts to mitigate the rigorous compliance with all inter-dependencies between program states when executing coverage-guided fuzzing strategies for enhancing the exploration power of all seeds. To this end, for a given target program, we first derive its control flow graph and identify the conditional instruction in each basic block and all the instructions affecting

it in the intermediate representation (IR) level [81, 126]. Then we relocate such instructions to their farthest dominator while preserving the original semantics of the conditional instruction, i.e., reducing program dependencies, as forming a "phantom" program. Next, *MirageFuzz* performs dual fuzzing, i.e., fuzzing the original program and its phantom program simultaneously, namely *source fuzzing* and *phantom fuzzing*. More specifically, after the *source fuzzing* upon a given seed S, we collect the unexplored program branches adjoining the explored program states and search for any seed generated by the *phantom fuzzing* which can be executed to explore any of such branches. If such a seed S' exists, we then identify the byte offset of S corresponding to the conditional instruction of the unexplored branch via taint analysis [127] and further update it using the corresponding condition value derived by S' to form a new seed for further *source fuzzing*. Eventually, executing the resulting new seed can advance the exploration of the program states bounded by the corresponding conditional instruction and thus enhances the seed effectiveness on exploring program states.

To evaluate the effectiveness of *MirageFuzz*, we first collected 18 real-world projects which were frequently adopted in recent fuzzing research as our benchmark suite. We further collected nine open-source coverage-guided fuzzers as our baselines for performance comparison with *MirageFuzz*. Our evaluation results suggest that *MirageFuzz* outperformed the baseline fuzzers significantly by 13.42% to 77.96% on average in terms of the edge coverage. Moreover, *MirageFuzz* exposed 29 previously unknown bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

5.1 Motivation

In this section, we use a sample code snippet following prior work as in Figure 5.1(a) [88] to motivate *MirageFuzz*. Specifically, the function `Origin` takes a character array `user` as input and processes it in nested branches. Note that many existing coverage-guided fuzzers [171, 49, 11, 94, 86, 159] incrementally increase code coverage under each iterative execution. Therefore, to trigger the crash on line 7 of function `Origin`, first, given an initial seed successfully exploring line 3, it is ideal to generate a mutant which can be executed to successfully explore line 4 under controllable effort with the resulting mutant retained as the new seed. The above operation is then repeated for the subsequent statements until line 6 can be successfully explored. However, the mutation space for each statement is essentially vast, e.g., for line 3, `user[0]` can be assigned with 256 possible values while it has to be `'M'` only to successfully access its scope. Thus, We can derive that the chance of a seed to explore 4 consecutive similar statements could be rather trivial, resulting in many underused seeds for fuzzing. To summarize, the exploration power on program states of a seed can be somewhat limited by applying many existing coverage-guided fuzzing strategies.

We consider the key factor limiting the effectiveness of coverage-guided fuzzing

```
1  void Origin(char *user){
2    user[4] = '\0';
3    if (user[0] == 'M') {
4      if (user[1] == 'A')
5        if (user[2] == 'Z')
6          if (user[3] == 'E')
7          // crash
8    } else {
9      ...
10     // other code
11   }
12   return;
13 }
14
```

(a) The original code

```
1    void Phantom(char *user){
2      user[4] = '\0';
3      if (user[0] == 'M');
4      if (user[1] == 'A');
5      if (user[2] == 'Z');
6      if (user[3] == 'E')
7        // crash
8      if (user[0] != 'M'){
9       ...
10      // other code
11     }
12     return;
13   }
14
```

(b) The phantom code

Figure 5.1: A motivation example code for *MirageFuzz*

strategies is that they require the seeds to rigorously abide by the program dependencies, i.e., being thoroughly executed, until exposing a bug/vulnerability. Specifically in Figure 5.1(a), the execution on one seed has to satisfy all the dependencies of line 7, i.e., lines 3 to 6, before exposing the relevant crash. Moreover, the program states subject to such program dependencies even have to be repeatedly accessed under iterative executions, e.g., line 3 has to be explored by all the iterative executions until exploring line 7. Therefore, in this chapter, we attempt to enhance the effectiveness of coverage-guided fuzzing strategies by mitigating the rigorous compliance with all inter-dependencies between program states on performing coverage-guided fuzzing strategies. In particular, a straightforward insight is to reduce program dependencies for preventing the aforementioned executions. We can observe from Figure 5.1(a) that actually the conditional statements of lines 3 to 6 are not related to one another, i.e., each of them can be satisfied independently (the operands of line 6 are irrelevant with line 3). Therefore, it is unnecessary to form nested dependencies among such conditional statements. Instead, we could reduce their dependencies as in function `Phantom` of Figure 5.1(b) where their respective executions are independent from each other. For example, the executions on all the seeds can easily access line 6 to check whether the runtime value of `user[3]` is `'E'`. Thus, we can infer that the chance to expose the crash in line 7 can be significantly enhanced compared with Figure 5.1(a). Such an example can be rather inspiring for how to enhance the power of exploring program states of the mutants. Specifically, suppose a fuzzing campaign is halted upon line 6 in Figure 1a, i.e., it satisfies line 5 but fails to satisfy line 6. We can attempt to obtain the byte offset of the running seed corresponding to line 6, i.e., the branch condition "`user[3] == 'E'`", via taint analysis. If we could also identify a seed which can be executed to explore the same branch condition in Figure 1b, we could then apply taint analysis again on that seed to obtain the operand value and use it to update the byte offset of the seed running in Figure 1a. At last, the resulting mutant can be executed to satisfy line 6 in Figure 1a and thus trigger the crash, indicating that the power of exploring program states of the original seed is improved. Accordingly, in this chapter, we are inspired to propose a technique which aims at reducing program dependencies for enhancing the

exploration power of seeds on program states.

Note that our mission in Figure 5.1 is seemingly close to advance the exploration of program states which many existing fuzzers [167, 134, 77, 24, 88, 22] attempt to tackle by proposing diverse techniques, e.g., recording the auxiliary states for program exploration depth or integrating constraint solver [35]. However, due to the aforementioned limitation of the well-adopted coverage guidance, they still generate massive ineffective seeds, i.e., only limited seeds are effective to explore program states. To illustrate, that essentially is the issue we attempt to address in this chapter.

5.2 Approach

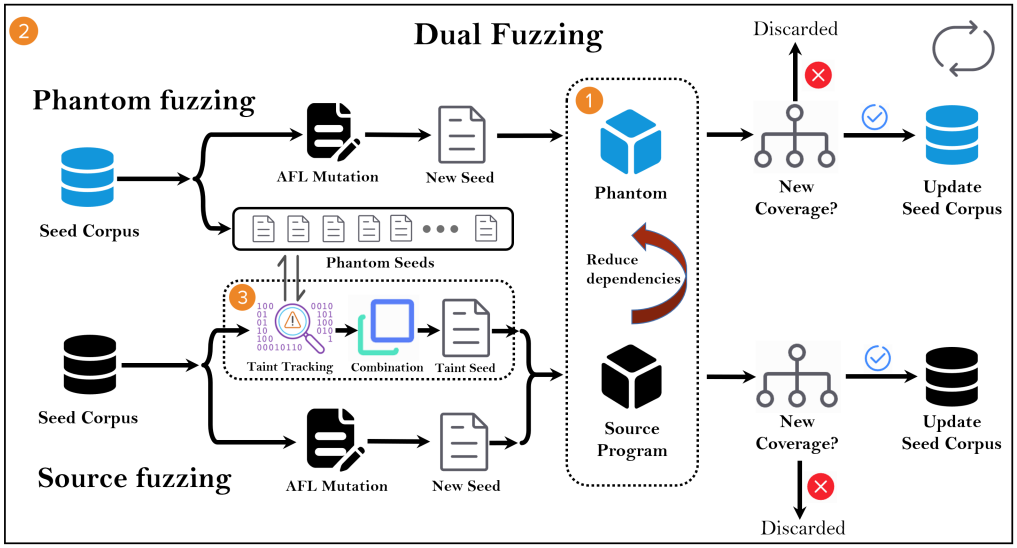


Figure 5.2: The workflow of *MirageFuzz*

Figure 5.2 shows the overall workflow of *MirageFuzz* which consists of three components. First, *MirageFuzz* creates a phantom program to reduce dependencies in the target program via a dependency reduction mechanism (marked as ❶ in Figure 5.2, Section 5.2.1). Next, *MirageFuzz* performs dual fuzzing—the *source fuzzing* to fuzz the original program and the *phantom fuzzing* to fuzz its corresponding phantom program simultaneously (❷, Section 5.2.2). Specifically, during iterative executions of the *source fuzzing* under a given seed S, *MirageFuzz* obtains the unexplored program branches adjoining the explored program states and searches for whether any of them has been explored by a seed (or multiple seeds) generated from the *phantom fuzzing*. At last, if such a seed S' exists, we then update the corresponding branch condition of S with the value derived by S' to form a new seed for future *source fuzzing* (❸, Section 5.2.3).

5.2.1 Dependency Reduction Mechanism

We first derive the control-flow graph (CFG) of the given target program and then identify all the branch instructions in the intermediate representation (IR) level [81, 126]. Next, for the conditional instruction in each basic block and all the instructions affecting it, we attempt to relocate them to their farthest dominator (in this paper, we follow prior work [39] that in a control-flow graph, a block a is a dominator of a block b if every path from the entry block to b must go through a). In this way, we essentially reduce the dependencies among program branches. Here we use the code snippets from a real-world project *jhead* [96] in Figure 5.3 with the CFG generated by LLVM *dot-cfg* pass [34] for illustration. In particular, Figure 5.3 presents a total of 8 basic blocks where `T` represents that the corresponding condition is evaluated as "true" and `F` represents otherwise. For instance, by relocating the conditional instruction `bool cmp3 = !strcmp(arg, "-proc")` of block ④ in its farthest dominator, i.e., entry block ①, their original dependency can thus be reduced. As a result, the execution on a seed can directly explore block ④ without exploring its original dependency with entry block ① in advance. Obviously, the chance to explore block ④ can be increased, indicating the exploration power of seeds can be increased.

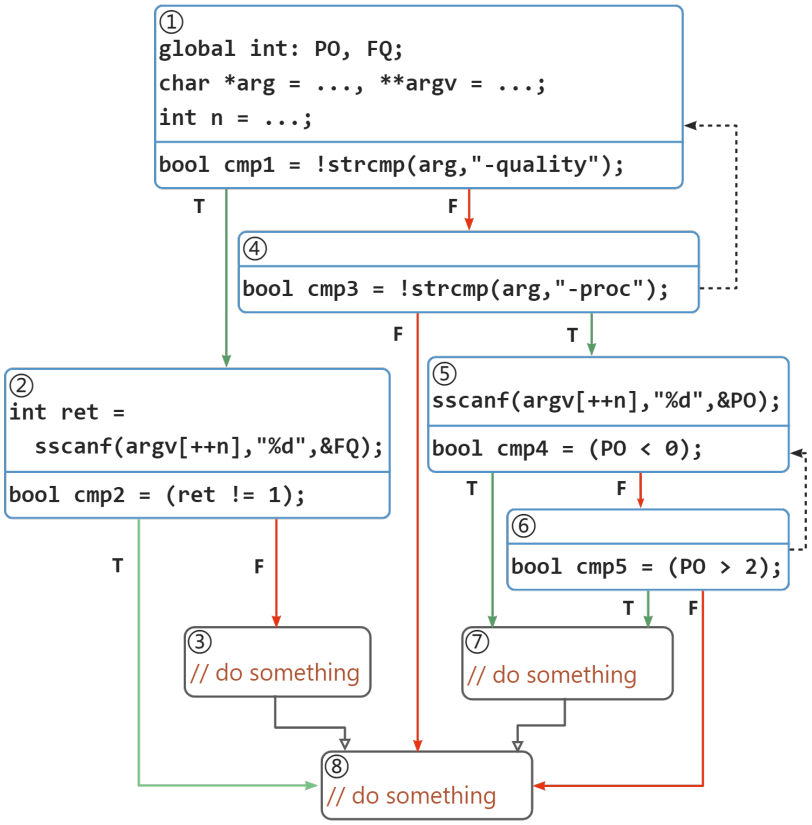


Figure 5.3: A simplified real-world example from *jhead*

Inspired by Section 5.1, we realize that by preserving the entry condition of each

Algorithm 3: Dependency Reduction Mechanism

```
1  Function ReduceDependencies(source, phantom):
2      blocks ← getBasicBlocks(source);
3      for each block in blocks do
4          branchCon ← getBranchExpression(block);
5          collectedBlocks ← slicingBasicBlocks(branchCon);
6          remainBlocks ← {b ∈ collectedBlocks || violateSemanticsAfterRelocating(b) is
             True ∧ isDominator(b, branchCon) is True};
7          if remainBlocks is ∅ then
8              entrance ← getEntryBlock(blocks);
9          else if entrance ← randomChoice(remainBlocks);
10         for each B in remainBlocks do
11             entrance ← entrance dominates B ? B : entrance;
12         then
13         instructions ← getRelatedInstructions(collectedBlocks);
14         move instructions to entrance;
15     phantom ← reconstruct(blocks);
16     return phantom;
```

branch in the target program, we can utilize the dependency-reduced program to facilitate the exploration of new program states in the original program. However, relocating instructions can easily violate the semantics of the original program branches (i.e., the entry condition of a program branch). For instance in Figure 5.3, relocating instructions from block ⑤ to entry block ① can violate the original semantics of block ③. In particular, by executing the instruction `sscanf(argv[++n], "%d", &P0)` of block ⑤ in entry block ① after instruction relocation, the value of variable `ret` calculated via instruction `sscanf(argv[++n], "%d", &FQ)` of block ② is changed since its variable `n` has already been updated in entry block ①. Accordingly, the semantics of condition `bool cmp2 = (ret != 1)` for block ③ would be violated. Note that while it is essential to preserve the semantics of branch conditions for correctly exploring their covered program states by executing seeds, it is unnecessary to preserve the semantics of other statements since changing them exerts no impact in accessing the updated blocks (i.e., for an operand not in a branch condition, its value is allowed to be changed after branch relocation if it is irrelevant to the operand value(s) of any branch condition).

In this chapter, we propose a dependency reduction mechanism to reduce program dependencies by relocating conditional instruction and the instructions affecting it while preserving original semantics of each conditional instruction as shown in Algorithm 3. For a given CFG of the target program, we first obtain all the basic blocks (represented as `blocks`) with the conditional instruction of each basic block (represented as `branchCon`, lines 2 to 4). Next, for each `block`, we identify all the basic blocks with the instructions affecting its conditional instruction, i.e., sharing variable usage, via program slicing [151]. Note that for the function `slicingBasicBlocks`, we perform the program slicing which extracts the dependent instructions of `branchCon` by applying

the use-def chains in both LLVM IR [81] and Memory SSA [103, 98] to obtain their associated blocks `collectedBlocks` (line 5). Furthermore, we filter out the dominators of `branchCon` whose semantics can be possibly violated by relocating the conditional instruction and the instructions affecting it (line 6). If there is no remaining basic block after filtering, we can infer that all the instructions can be relocated in the entry block (lines 7 to 8). Otherwise, we identify the farthest dominator in CFG to which all the instructions can be relocated without violating semantics on the conditional instruction (lines 10 to 12) and perform the instruction relocation (lines 13 to 14). After the iterative executions on all the collected conditional instructions, we obtain a phantom program for *MirageFuzz* (line 15). Note that for the phantom program, we only preserve semantics for conditional instructions as in the original program such that its adopted seeds can be used to advance program state exploration when fuzzing the original program (illustrated later) while the semantics of the rest instructions does not matter for building our phantom program. Accordingly in Figure 5.3, the conditional instructions located in ② and ⑤ cannot be relocated since `++n` violates the semantics of the branch conditions where `n` is an operand. On the contrary, the conditional instructions in ⑥ and ④ are relocated to ⑤ and ① since the semantics of all involved branch conditions can be preserved after relocation (more details are presented in our *GitHub* page [53]).

5.2.2 Dual Fuzzing

Given the phantom program by applying the dependency reduction mechanism, *MirageFuzz* performs dual fuzzing, i.e., the *source fuzzing* to fuzz the original program and the *phantom fuzzing* to fuzz the phantom program simultaneously, under the identical initial seed corpus and execution time budget. During the *source fuzzing*, *MirageFuzz* collects the unexplored program branches adjoining the explored program states. Then, *MirageFuzz* searches for any seed executed to explore such collected branches in the *phantom fuzzing* for later generating new seeds to advance the *source fuzzing*.

Algorithm 4 presents the details for the dual fuzzing. In particular, for the *source fuzzing*, we first adopt the AFL mutation strategy [159] to generate mutants out of our initial seed corpus (lines 2 to 5). If any mutant is executed to explore new edges, it is added into the seed corpus for further iterative executions (lines 6 to 7). Meanwhile, we derive the unexplored program edges adjoining the explored program states by executing the given mutant, and check whether they can be explored by executing any of the seeds generated from the *phantom fuzzing* (lines 8 to 9). If such a seed exists, we then adopt the taint-based mutation mechanism (illustrated later) to generate a new seed for the future *source fuzzing* (lines 10 to 13), and if they explore new edges, they are added into the seed corpus for next iteration. Eventually, after total time budget `budget` is consumed, *source fuzzing* terminates and returns `None` (line 14).

For the *phantom fuzzing*, we create a dictionary `edgeDic` to store the information of the explored edges with their corresponding executed seeds (lines 16 to 17). For each iterative execution, we first check the real-time unexplored edges from the *source*

Algorithm 4: Dual Fuzzing

```
1  Function FuzzingSourceProgram(initialSeed, budget):
2      seeds ← {initialSeed} ;
3      while fuzzing time not exceed budget do
4          for each seed in seeds do
5              mutant ← AFLMutation(seed);
6              if mutant has new edges then
7                  seeds ← seeds ∪ {mutant};
8              edges ← getUnexploredEdges(seed);
9              phantomSeeds ← requestSeedsFromPhantom(edges);
10             taintSeeds ← generateSeedWithTaint(seed, phantomSeeds);
11             for each tSeed in taintSeeds do
12                 if tSeed has new edges then
13                     seeds ← seeds ∪ {tSeed};
14     return None
15 Function FuzzingPhantomProgram(initialSeed, budget):
16     seeds ← {initialSeed};
17     edgeDic ← [[edge ⇒ {}]];
18     while fuzzing time not exceed budget do
19         if has requests from source fuzzing then
20             resp ← {};
21             for each edge in requests do
22                 resp ← resp ∪ edgeDic[[edge]];
23             Send resp to source fuzzing;
24         for each seed in seeds do
25             mutant ← AFLMutation(seed);
26             if mutant has new edges then
27                 seeds ← seeds ∪ {mutant};
28                 for each edge in mutant do
29                     edgeDic[[edge]] ← edgeDic[[edge]] ∪ {mutant};
30     return None;
```

fuzzing (lines 18 to 19). Then, we iterate each unexplored edge to find whether it has been already explored by executing any seed generated by the *phantom fuzzing* (lines 20 to 23). Similar to the *source fuzzing*, we also adopt the AFL mutation strategy to generate mutants (lines 24 to 25). If executing any mutant explores new edges, it is added to the seed corpus where `edgeDic` is updated (lines 28 to 29).

In Figure 5.3, assume a `seed` is executed to explore the path [①,④,⑤,⑦,⑧] for the *source fuzzing*. Then we can derive the unexplored edges adjoining the explored path, i.e., ①→②, ④→⑧, and ⑤→⑥, which are further collected in `edgeDic` for the *phantom fuzzing*. For each edge, *MirageFuzz* searches for any seed executed to explore it in the *phantom fuzzing*. At last, all the collected seeds from the *phantom fuzzing* are used to generate new seeds for future *source fuzzing* via the taint-based mutation mechanism.

Algorithm 5: Taint-Based Mutation Mechanism

```
1  Function generateSeedWithTaint(sourceSeed, phantomSeeds):
2      taintSeeds ← {};
3      exploredEdges ← getAllExploredEdgesFromSource();
4      for each mSeed in phantomSeeds do
5          for each edge in mSeed's execution path do
6              if edge not in exploredEdges then
7                  taintPos ← taintPosition(edge, sourceSeed);
8                  value ← taintContent(edge, mSeed);
9                  mutant ← sourceSeed;
10                 mutant[taintPos] ← value;
11                 taintSeeds ← taintSeeds ∪ {mutant};
12     return taintSeeds;
```

5.2.3 Taint-Based Mutation Mechanism

We develop the taint-based mutation mechanism to derive the byte offset corresponding to the conditional instruction of the given seed from the *source fuzzing* with its value derived by the *phantom fuzzing* via taint analysis [127]. Specifically, in our adopted taint analysis, the input stream (i.e., the seed) is referred to as the sole taint source. In order to trace the tainted labels at runtime, we define taint propagation rules to map the tainted input labels and output labels (e.g., `add`, `store`, and `load` instructions) at a particular level of the operation hierarchy. Accordingly, given a specific branch condition, we can collect its corresponding label for its operand or the relevant byte offset of such a operand in the seed.

Algorithm 5 illustrates the details of the taint-based mutation mechanism which is initialized with the seed to be mutated by the *source fuzzing* (denoted as `sourceSeed`) and the collected seeds from the *phantom fuzzing* which can be executed to explore the identified unexplored edges from the *source fuzzing* (denoted as `phantomSeeds`). We first obtain all real-time explored edges (line 3). Next, for each seed in the `phantomSeeds`, we iterate its explored edges (lines 4 to 5). If the edge is not explored by the *source fuzzing*, we then derive the byte offset `taintPos` corresponding to the conditional instruction by taint analysis in the `sourceSeed` (lines 6 to 7). Subsequently, we also obtain the corresponding condition `value` by taint analysis in the given `mSeed` from `phantomSeeds` (line 8). To illustrate, note that to prevent the misallignment between the byte offset of `mSeed` corresponding to the condition `value` and `taintPos`, we activate two taint analysis processes for `sourceSeed` and `mSeed` respectively. Specifically, we first apply taint analysis in the *source fuzzing* for obtaining the byte offset of the seed (i.e., `sourceSeed`) corresponding to the given unexplored branch condition from the source program. If we could identify a seed (i.e., `mSeed`) which can be executed to explore such a branch condition in the *phantom fuzzing*, we then apply taint analysis again on that seed to obtain values of the involved operand corresponding to the branch condition. Accordingly, we generate a mutant by updating `taintPos` with the corresponding `value` from the seed generated by the *phantom fuzzing*, and then store it

in the set `taintSeeds` (lines 9 to 11). At last, the resulting `taintSeeds` is used for advancing the future *source fuzzing*. We take the same seed S exploring path [①,④,⑤,⑦,⑧] mentioned in Section 5.2.2 as an example. Suppose we have another seed S' generated by the *phantom fuzzing* which has satisfied the conditional instruction for block ⑥. Next, by performing taint analysis on S, we identify its byte offset impacting the value of `PO` that determines the transition ⑤→⑥ or ⑤→⑦. We further figure out that the value of `PO` in S' is 14 via taint analysis on S'. Eventually, we replace the value of `PO` in S with 14 to generate a new seed for exploring the new edge ⑤→⑥ for the *source fuzzing*.

5.3 Implementation

We implement *MirageFuzz* using C/C++. Specifically, we perform instrumentation via LLVM pass [81] to obtain runtime information of target programs. Accordingly, we build *MirageFuzz* via the AFL implementation. Furthermore, we modify the taint analysis library *libdft* [74] to implement the taint-based mutation mechanism.

We encounter three main challenges when implementing *MirageFuzz*. First, it is challenging to identify the unexplored edges adjoining the explored program states via instrumentation. Second, adapting the existing taint analysis tool for the taint-based mutation mechanism in *MirageFuzz* potentially leads to non-negligible engineering effort. At last, implementing *phantom fuzzing* tends to cause unexpected crashes which terminate the execution on the phantom program early to prevent it from exploring deep program states. We then illustrate how we address the challenges as follows.

5.3.1 Instrumentation

Note that in the *source fuzzing*, we aim at recognizing unexplored edges adjoining the explored program states by executing a seed. To this end, we insert an observation instruction ahead of a given branch instruction to monitor whether any of its associated edges has been explored by observing the associated sink state of such branch. If the sink of the given branch instruction is not reached, it indicates that the corresponding edge is unexplored. Therefore, combining with the real-time collected explored edges, we can derive the unexplored edges adjoining them.

5.3.2 Dynamic Taint Analysis

We adopt *libdft* [74], a stable and efficient binary-level dynamic taint analysis framework adopted by many existing works [21, 167, 121], to implement the taint-based mutation mechanism. Although *libdft* implemented the taint propagation rules for 146 instructions, their default taint propagation rules still cannot cover our required instructions, e.g., `bswap` (reversing the byte order of a register) and `shl` (shifting the bits of a register to the left). We also analyze that multiple taint labels of instructions `movzx` and `movsx` can cause "over-taint" issues, leading to inefficient taint tracking. To tackle

these issues, we define our own taint propagation rules to cope with 11 new instructions and revoke the redundant taint labels for *libdft* to improve the taint-based mutation mechanism.

5.3.3 Crash Handling in *Phantom Fuzzing*

Generating the phantom program can inevitably devastate many dependencies of the original program, incurring crashes which potentially prevent the *phantom fuzzing* from exploring sufficient states of the phantom program. To address this issue, we design a "try-catch" mechanism to bypass these unexpected crashes. More specifically, we first capture all crash-related system signals and design their corresponding handler. Next, we obtain the runtime `program counter` [136] value, and increase it with the length of the real-time crash-triggered instruction to bypass it. As a result, *phantom fuzzing* can proceed to explore program states instead of being halted by the unexpected crashes.

With the solutions above, *MirageFuzz* is made scalable since it can be directly adopted upon any projects built upon LLVM-based compiler (e.g., *clang* [80]) without any additional adaptation effort.

5.4 Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness of *MirageFuzz* upon 18 benchmark programs compared with nine baseline fuzzers. In particular, we attempt to answer the following research questions:

- ***RQ1:*** Is *MirageFuzz* effective compared with the baseline fuzzers?
- ***RQ2:*** Is each component of *MirageFuzz* effective in terms of ablation study?

We also report and analyze the bugs on our benchmark suite exposed by *MirageFuzz*. Note that all source code of *MirageFuzz* and the evaluation details are presented in our *GitHub* pages [53, 2].

5.4.1 Baseline Fuzzers and Benchmark

Baseline fuzzers. To collect the baseline fuzzers for performance comparison with *MirageFuzz*, we determine to first select the coverage-guided fuzzers recently published in prestigious software engineering and security conferences, e.g., ICSE, FSE, S&P, and CCS. Next, we filter the selected fuzzers based on their source code availability and the feasibility of their execution environments. Eventually, we collect a total of nine fuzzers to form our baselines. More specifically, we select six coverage-guided fuzzers, i.e., the latest versions of AFL [171], AFL++ [49], LafIntel [79], $Havoc_{MAB}$ [159], MOPT [94] and FairFuzz [86]. Moreover, we also adopt three recent fuzzers with constraint solvers as our baselines, i.e., Angora [21], MEUZZ [24] and QSYM [167], to further compare the performance of our insight which enhances the exploration power of seeds without

leveraging the power of the constraint solver and the constraint-solving-based fuzzers on their well-performed benchmarks.

Benchmark. Following multiple prior works [94, 21, 167, 159, 86], we first construct our benchmark suite by collecting the projects commonly adopted by the fuzzers recently published in the aforementioned top software engineering and security conferences. Next, we also include 6 projects from FuzzBench [99] in our benchmark suite. As a result, our benchmark suite is formed by 18 frequently used projects with their latest versions. We also present the statistics of our adopted benchmarks in our *GitHub* page [2].

5.4.2 Environment Setup

Our evaluations are performed on a server with 64-core 2.80GHz Intel(R) Xeon(R) Gold 6342 CPUs and 64 GiB RAM running on 64-bit Linux version 4.15.0-172-generic Ubuntu 18.04.

Following many prior work [171, 49, 79, 86, 94, 159, 167], we set the total execution time budget to 24 hours. Meanwhile, all our evaluation results are averaged out of 10 runs. Furthermore, we follow the seed selection strategy in prior work [86, 76, 150, 60] to construct the initial seed corpus for each benchmark program from either its corresponding AFL seed collection or its own test suite.

In this chapter, we adopt edge coverage to represent code coverage, as all our studied baseline fuzzers [171, 49, 79, 159, 94, 86, 167]. Here an edge refers to a conditional jump between two basic blocks in the program control flow. Note that since *MirageFuzz* enables two instances in dual fuzzing, for fair performance comparison, we evaluate all our baseline fuzzers in a parallel fuzzing manner, i.e., enabling one additional instance which shares the same seed corpus during the fuzzing campaign for all the baseline fuzzers (except QSYM and MEUZZ which enable three processes sharing the same seed queue by default [167, 24]).

5.4.3 Result Analysis

RQ1: the Effectiveness of *MirageFuzz*

Table 5.1 presents the edge coverage results of our studied fuzzers upon our benchmark suite. Noticing that MEUZZ requires additional computation resources to analyze the target program for fuzzing, we mark a benchmark as **N/A** when MEUZZ fails to complete its execution after consuming all memory resources (e.g., *objcopy*). Overall, we can observe that *MirageFuzz* outperforms all other fuzzers significantly. In particular, *MirageFuzz* explores 5773 edges on average, which is 13.42% more than the top-performing baseline fuzzer QSYM (5090 explored edges) and 77.96% more than the worst-performing baseline fuzzer LafIntel (3244 explored edges) in our study. Additionally, *MirageFuzz* consistently outperforms all the baseline fuzzers upon each benchmark program. To illustrate the significance of the performance, we also adopt the

Table 5.1: Effectiveness of *MirageFuzz*

Benchmark	AFL	AFL++	LafIntel	FairFuzz	MOPT	$Havoc_{MAB}$	QSYM	MEUZZ	Angora	$MirageFuzz_{taint}$	$MirageFuzz_{splice}$	*MirageFuzz*
readelf	4511	7137	5523	8839	11537	12841	11880	11883	13228	11742	12126	**13611**
nm	2657	3770	3205	4091	5134	5924	6548	5724	4471	5528	5389	**7584**
objdump	2843	3551	3471	5753	5780	6007	6314	N/A	5265	5923	6174	**7735**
objcopy	3240	3146	2856	4080	7456	7528	8167	N/A	9830	7379	8400	**10473**
size	2095	2185	2510	3447	3622	3651	4973	4206	5038	4011	3999	**5469**
jhead	161	177	199	161	165	169	784	804	176	753	767	**858**
pcre2	6027	6446	6510	6460	6209	6566	6277	6516	4082	6178	6555	**6714**
pngfix	1044	1043	1835	1033	1152	1045	1988	2093	1774	1761	1778	**2195**
strip	3022	3606	3104	4225	6970	6927	6725	N/A	9071	7027	7357	**9417**
listaction_d	3770	4118	4046	4164	4042	3831	4101	3889	3391	4097	3914	**4194**
libxml2	9657	9015	4468	8085	9559	9268	10346	N/A	7053	9019	8991	**10992**
libpng	863	861	2057	1060	1061	868	2021	N/A	1872	1873	2199	**2313**
re2	6911	**6915**	6874	6874	6855	6844	6837	N/A	6143	6460	6877	6892
swftocxx	4519	4544	4369	4378	4327	4413	4483	4545	4474	4359	4551	**4675**
jsoncpp	1010	1010	1002	1012	1010	1010	946	N/A	886	987	1001	**1023**
lcms2	888	1395	935	864	864	862	2631	N/A	1083	879	1035	**2824**
file	1268	1452	852	1414	1551	1480	1857	1861	1627	1493	1396	**1970**
libjpeg-turbo	4213	4193	4573	3281	4180	4623	4745	N/A	3797	4380	4658	**4973**
average	3261	3587	3244	3846	4526	4659	5090	4613	4626	4658	4826	**5773**
p-value	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006	-

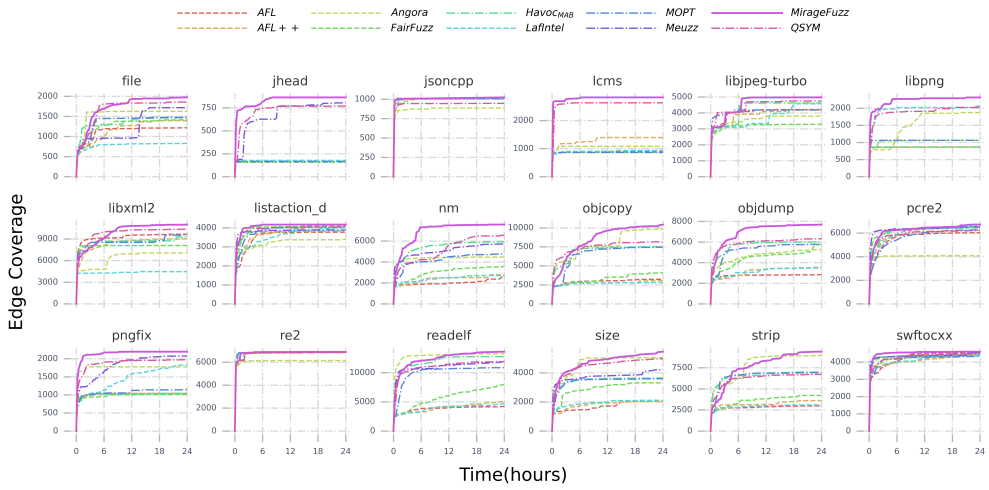


Figure 5.4: The edge exploration trends of all fuzzers

Mann-Whitney U test in our evaluation. We can observe that in Table 5.1 where the p-value of *MirageFuzz* comparing with other studied fuzzers in terms of the average edge coverage are all far below 0.05, which indicates that *MirageFuzz* outperforms all selected fuzzers significantly ($p < 0.05$). Furthermore, Figure 5.4 presents the edge coverage trends of all our studied fuzzers upon each benchmark program within the 24-hour execution. We can observe that *MirageFuzz* dominates the baseline fuzzers under most of the execution time. Such results altogether indicate that *MirageFuzz* is a rather powerful coverage-guided fuzzer.

We also investigate the effectiveness of exploring unique edges (i.e., edges that can only be explored by a given fuzzer) for all our studied fuzzers. In our evaluation, *MirageFuzz* can achieve the best performance by exploring 4268 unique edges on top of the whole benchmark suite averagely, which outperforms the top-performing baseline Angora by 62.16% (4268 vs. 2632 edges). Due to the page limit, we present the performance details in our GitHub page [2].

Finding 1: MirageFuzz is a rather powerful coverage-guided fuzzer which can significantly and consistently outperform the adopted baseline fuzzers.

Interestingly, while QSYM and Angora are generally more effective than other baseline fuzzers, the fact that *MirageFuzz* significantly outperforms them on all benchmark programs without applying a constraint solver indicates that its insight which enhances the exploration power of seeds via dual fuzzing only is potentially even more powerful in exploring program states.

Finding 2: The mechanisms adopted by MirageFuzz are potentially more effective than applying constraint solver for exploring program states.

RQ2: the Effectiveness of Different Components in *MirageFuzz*

To further understand the mechanism adopted by *MirageFuzz*, in this section, we perform in-depth ablation studies to investigate the effectiveness of the dedicated components designed for *MirageFuzz*, i.e., the *phantom fuzzing* and the taint-based mutation mechanism.

Effectiveness of the phantom fuzzing. Investigating the effectiveness of the *phantom fuzzing* for *MirageFuzz* is essentially equivalent to investigating the effectiveness of using the condition value derived by the *phantom fuzzing* for mutating the corresponding condition of the given seed for the *source fuzzing*. Accordingly, we determine to create a technique variant $MirageFuzz_{taint}$ of *MirageFuzz* which tracks the byte offset impacting the unexplored condition of a given seed and then applies random mutation on the corresponding byte offset. Meanwhile, we activate another *source fuzzing* process to replace the original *phantom fuzzing* process.

Table 5.1 also presents the edge coverage results of $MirageFuzz_{taint}$. We can observe that *MirageFuzz* significantly outperforms $MirageFuzz_{taint}$ by 23.94%. Moreover, we can also find that both $Havoc_{MAB}$ and QSYM outperform $MirageFuzz_{taint}$ by 0.02% and 9.27% respectively. Such results suggest that *phantom fuzzing* is essential in strengthening the effectiveness of *MirageFuzz*.

Finding 3: The phantom fuzzing is critical for MirageFuzz to augment its edge coverage performance.

Effectiveness of taint-based mutation mechanism. We create a technique variant $MirageFuzz_{splice}$ which replaces the taint-based mutation mechanism by randomly identifying a byte offset of a given seed in the *source fuzzing* and splicing the given seed and a randomly selected seed for the *phantom fuzzing* at the identified byte offset to generate a mutant for the *source fuzzing*.

Table 5.1 also presents the edge coverage results of $MirageFuzz_{splice}$ where *MirageFuzz* outperforms $MirageFuzz_{splice}$ by 19.62%. Such a result clearly demonstrates that applying the taint-based mutation mechanism can advance the effectiveness of the *phantom fuzzing* by precisely positioning the byte offset associated with the unexplored condition and providing the condition value to generate a mutant which can be executed to facilitate the *source fuzzing*.

> *Finding 4: The taint-based mutation mechanism is essential for MirageFuzz in facilitating its fuzzing efficacy.*

We further investigate the taint-analysis time in our fuzzing campaign (presented in our *GitHub* page [2]), where it ranges from 1745 to 33589 seconds averagely during 24-hour runs. Notably, even though it costs 33589 seconds for taint analysis on project `strip`, *MirageFuzz* still achieves the best edge coverage (i.e., covering 9417 edges) averagely in 24-hour run.

5.4.4 Bug Report and Analysis

In this chapter, we obtain all the crashes and then manually identify the buggy location through stack tracing and analyze their respectively causes. Accordingly, we derive unique bugs via debugging. We then report our exposed bugs to the developers with the essential information that can help them generate a patch. Overall, applying *MirageFuzz* exposes 29 previously unknown bugs upon our benchmark suite where 7 were confirmed and 6 were fixed by the corresponding developers. Meanwhile, AFL, AFL++ and MEUZZ detect 2 out-of-memory bugs in project *swftocxx*, and AFL++, FairFuzz and QSYM expose 2 heap-buffer-overflow bugs in project *listaction_d*. Note that *MirageFuzz* can expose all the bugs exposed by all other fuzzers. We illustrate all our bug types, e.g., a use-of-uninitialized-value bug refers to using a variable without initialization, in our *GitHub* pages [2]. Table 5.2 presents the details of the previous unknown bugs exposed by *MirageFuzz*.

Infinite Loop in *pcre2test*

We have reported a bug on project *pcre2* [115]—a set of C functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5 [117]. It was assigned with an issue ID 141 [116] and has been confirmed and fixed by developers. This bug was exposed by running *pcre2test*, one of the executable programs in project *pcre2* with the specified input files only generated by *MirageFuzz*.

While processing the input files, an infinite loop in function `process_data(void)` occurred as shown in Figure 5.5.

For the `while` condition `needlen >= dbf_size` and the loop body `dbf_size *= 2`, we analyze that the value of `needlen` potentially incurs infinite looping due to a

Table 5.2: The bug information

Program	Bug Type	Number	Status
pcre2	Infinite loop	1	*confirmed and fixed*
nm	Infinite loop	1	*reported*
jhead	Use-of-uninitialized-value	3	*confirmed and fixed*
strip	Out-of-memory	1	*confirmed and fixed*
pngfix	Use-of-uninitialized-value Infinite loop	1 1	*confirmed and fixed* *confirmed*
listaction_d	Segmentation fault Heap-buffer-overflow	6 3	*reported* *reported*
swftocxx	Segmentation fault Heap-buffer-overflow Allocation-size-too-big Out-of-memory	5 4 1 2	*reported* *reported* *reported* *reported*

```
int process_data(void)
{
  // ...
  // p is a section from input file, li is s64, i is s32, needlen and dbuffer_size
   are u64.
  li = strtol((const char *)p, &endptr, 10);
  if (S32OVERFLOW(li)) { return OK; }
  i = (int32_t)li;
  if (i-- == 0) { return OK; }
  // ...
  replen = CAST8VAR(q) - start_rep;
  needlen += replen * i;

  if (needlen >= dbuffer_size)
  {
   // ...
   while (needlen >= dbuffer_size)
     dbuffer_size *= 2;
   // ...
  }
  // ...
 }

```

Figure 5.5: Infinite loop in *pcre2test*.

possible integer overflow. In fact, one of our input files sets `i = -10`, which in turn assigns `needlen` with the value resulting in an infinite loop.

Correspondingly, the developers made a simple fix, i.e., patching `i-- == 0` as `i-- <= 0`. They commented on this bug as follows:

> *"A negative repeat value in a pcre2test subject line was not being diagnosed, leading to infinite looping."*

Use-of-Uninitialized-Value in *pngfix*

We reported a use-of-uninitialized-value bug in project *libpng* [92] only exposed by *MirageFuzz* under the instrumentation by MemorySantizer [139]. In particular, the bug

was exposed by running the generated seed from *pngfix*, one of the executable programs in project *libpng*, confirmed with the *GitHub* issue ID 424 [118] and fixed later.

The buggy code snippet is presented in Figure 5.6 where the uninitialized value reported by Memory sanitizer comes from `png_ptr->big_row_buf` and `png_ptr->big_prev_row`.

```
1  void png_read_start_row(png_structrp png_ptr)
2  {
3     // ...
4     if (png_ptr->interlaced != 0)
5       png_ptr->big_row_buf = (png_bytep)
6        png_calloc(png_ptr,row_bytes+48);
7     else
8       png_ptr->big_row_buf = (png_bytep)
9        png_malloc(png_ptr,row_bytes+48);
10    png_ptr->big_prev_row = (png_bytep)
11      png_malloc(png_ptr,row_bytes+48);
12    // ...
13 }
14
```

Figure 5.6: Use-of-uninitialized-value in *pngfix*.

The developers believed that this problem was caused by lacking the memory initialization before using the memory requested by `malloc` and then fixed the bug by invoking `memset` in the end of the code snippet in Figure 5.6 with the following feedback:

> *"In my opinion it is due to the fact that `png_malloc` just calls `malloc` but doesn't initialize the memory. I can work on that and improve it. It would really help to avoid similar issues in the future. "*

Use-of-Uninitialized-Value in *jhead*

We reported multiple use-of-uninitialized-value bugs of project *jhead*. These bugs, reported in a GitHub issue (ID 53) [69], were confirmed and fixed.

The relevant buggy code snippet in function `ReadJpegSections` is shown in Figure 5.7, where `Data` is a pointer to an allocated heap memory segment by invoking `malloc`. However, such a memory segment is not initialized before `Data` is used in the subsequent procedure, and thus leads to a vulnerability.

Eventually, the developer generated a patch by invoking `memset` to initialize the value of the related memory after it is allocated.

> *"Or at least that should fix it. ..., but I could see how this could be triggered."*

Out-of-Memory in *strip*

We have reported one out-of-memory bug as a bugzilla issue with ID 29495 [140] when executing project *strip*, which was confirmed and fixed by the associated developers.

```
int ReadJpegSections (FILE * infile, ReadMode_t ReadMode)
 {
  // ...
  uchar * Data;
  // ...
  Data = (uchar *)malloc(itemlen+20);
  if (Data == NULL){
    ErrFatal("Could not allocate memory");
  }
  Sections[SectionsRead].Data = Data;
  // ...
 }

```

Figure 5.7: Use-of-uninitialized-value in *jhead*.

The function `exif.c:rewrite_elf_program_header` in Figure 5.8 reveals the relevant buggy code snippet. By using the input generated by our approach, the execution on *strip* keeps consuming memory and causes an out-of-memory bug. In our evaluation, *strip* consumes 64 GiB memory in our server in about two minutes.

Similar to `malloc`, we found that `bfd_zalloc` is a function that allocates memory in the heap, located in the loop in line 18. The loop only terminates by updating `isec` surrounded by a conditional code region (lines 8 to 12). Therefore, an out-of-memory bug is triggered if *strip* fails to enter such code region, i.e., the condition of such a code region cannot be satisfied.

```
  static bool rewrite_elf_program_header
    (bfd *ibfd, bfd *obfd, bfd_vma maxpagesize)
  {
    // ...
    isec = 0;
    do {
      // ...
      if (IS_CONTAINED_BY_LMA(output_section, segment, map->p_paddr, opb) ||
        IS_COREFILE_NOTE(segment, section)) {
        // ...
        ++isec;
      }

      // ...
      if (isec < section_count) {
        // ...
        // bfd_zalloc allocates memory.
        map = (struct elf_segment_map *) bfd_zalloc(obfd, amt);
        // ...
      }
      continue;
    } while (isec < section_count);
    // ...
  }

```

Figure 5.8: Out-of-memory in *strip*.

The developers fixed this bug by refactoring the whole function to avoid memory overflow. They also commented the bug as follows:

"It's important that the later tests not be more restrictive. If they are it can lead to the situation triggered by the testcases, where a section seemingly didn't fit and thus needed a new mapping. It didn't fit the new mapping either, and this repeated until memory exhausted."

5.5 Limitations

MirageFuzz has two limitations. Firstly, similar to many existing fuzzers, *MirageFuzz* necessitates intervention in the compilation process of the target project, thereby requiring access to the project source code for fuzzing initiation. Consequently, in scenarios where source code acquisition is unfeasible, *MirageFuzz* may fail to yield the anticipated results. Secondly, *MirageFuzz* is implemented as an LLVM pass, presently only compatible with projects compiled using LLVM. For projects reliant on alternative compilation tools, additional adaptation is imperative for *MirageFuzz* to function effectively. In light of these constraints, future endeavors aim to reimplement *MirageFuzz* at the assembly code level, thereby decoupling it from higher-level programming languages and consequently enhancing its versatility and applicability.

5.6 Summary

In this chapter, we propose the concept of *phantom program*, which is built to mitigate the over-compliance of program dependencies to enhance the exploration capacity of all seeds. Accordingly, we build a coverage-guided fuzzer namely *MirageFuzz* which performs dual fuzzing for the original program and the phantom program simultaneously and adopts the taint-based mutation mechanism to generate new mutants by combining the resulting seeds from dual fuzzing via taint analysis. To evaluate the effectiveness of *MirageFuzz*, we select 18 frequently used projects to form our benchmark suite and nine popular open source fuzzers to form our baseline fuzzers. The evaluation results show that *MirageFuzz* outperforms the baseline fuzzers from 13.42% to 77.96% in terms of edge coverage averagely in our benchmark. *MirageFuzz* also exposes 29 previously unknown unique bugs where 7 of them have been confirmed and 6 have been fixed by the corresponding developers.

Chapter 6

JITfuzz: Coverage-Guided Fuzzing for JIT in JVM

Java Virtual Machine (JVM) has been widely adopted in many popular application domains, e.g., mobile applications and cloud computing, by supporting the execution of Java bytecode compiled from various high-level programming languages, e.g., Java, Scala, and Clojure [93]. However, while JVM is advanced in adopting interpretation in addition to compilation for cross-platform execution, interpreting JVM-based programs incurs high performance overhead. To tackle this issue, the Just-in-Time compiler (JIT) has been designed to improve runtime compilation performance of JVM-based programs by compiling selected code (e.g., frequently executed code) into native machine code. In this way, the resulting JVM bytecode can be executed directly without the costly interpretation process, leading to efficient program execution. To date, JIT has become a crucial component in JVM implementations where its correctness plays a vital role in ensuring correct and efficient execution of JVM-based programs.

While it is evident that testing JITs to ensure their correctness is vital for the correct execution for JVMs, how to effectively and efficiently test JITs remains rather challenging due to the following reasons. First, JITs can hardly be thoroughly tested because they include diverse optimization techniques which are activated under diverse scenarios. That said, to thoroughly test JITs, it is essential to create as many such optimization scenarios as possible, which can be potentially challenging. Second, while random/probabilistic mutation becomes a major paradigm adopted by many fuzzers [171, 11, 94, 86], it is nevertheless inefficient for JIT testing since massive resulting mutants cannot conform to JVM specification and executing them easily terminates JIT testing early (e.g., in the input verification phase [145]) to prevent testing deep JIT states. At last, although traditional JVM fuzzing techniques have proven that applying control-flow mutators can improve testing effectiveness, such mutators can hardly be directly applied in fuzzing JITs, e.g., adding new transitions in the existing control flow graphs can easily break variable dependencies [26], causing early-terminated JVM executions and thus leading to insufficient JIT testing. Therefore, albeit general-purpose JVM

fuzzing techniques can occasionally expose JIT bugs, there still is a pressing need for a dedicated fuzzing technique specifically targeting JITs.

In this chapter, we propose *JITfuzz*, a coverage-guided fuzzing framework specifically targeting JITs. In particular, testing JITs essentially is testing their mechanisms for compiling bytecode into native machine code during runtime. Intuitively, such mechanisms can expose *interesting* behaviors (which may expose bugs) when triggering the usage of their optimization techniques and enriching the control flows of target JITs. Therefore in this chapter, after explicitly launching JITs via specific JVM commands, *JITfuzz* adopts two types of mutators to advance thorough testing of the JIT mechanism. More specifically, optimization-activating mutators are proposed to trigger the activation of typical optimization techniques, e.g., the function-inlining-activating mutator is applied to create scenarios where the function inlining strategy [62] is activated. Meanwhile, noticing that mutating program control flows can potentially affect JIT optimizations, we also propose control-flow-enriching mutators to *safely* complicate program control flows of the generated test programs (i.e., alleviating early termination). Furthermore, *JITfuzz* adopts a mutator scheduler to dynamically schedule the mutators to optimize the runtime testing coverage of JITs.

To evaluate the effectiveness of *JITfuzz*, we first construct a real-world benchmark suite composed of 6 commonly adopted projects from prior JVM fuzzing work [26, 176] and 10 popular open-source projects in *GitHub* [52]. Next, by choosing OpenJDK19 [108] as our target JVM, we conduct a set of experiments to explore the effectiveness of *JITfuzz* and its components. The evaluation results suggest that *JITfuzz* outperforms state-of-the-art mutation-based JVM fuzzer *Classming* [26] and generation-based JVM fuzzer *JavaTailor* [176] by 27.9% and 18.6% respectively in terms of the code coverage. Meanwhile, all our proposed technical components in *JITfuzz*, including the mutators and mutator scheduler, are effective. For instance, adopting mutator scheduler can increase the code coverage by 10.2% on average. Moreover, *JITfuzz* successfully detects 36 previously unknown bugs on three commercial JVMs while none of them can be detected by *Classming* or *JavaTailor*. Specifically, 27 bugs have been confirmed by the corresponding developers and 16 have already been fixed. Moreover, 23 of them are JIT bugs where 18 have been confirmed and 7 have been fixed.

6.1 Motivation

JIT has been demonstrated to significantly impact the runtime performance of JVM-based applications and thus strongly recommended by industrial developers. While testing JITs to ensure their correctness is essential, there exists no dedicated testing technique for such specific purpose to our best knowledge. Albeit the existing general-purpose JVM fuzzers [28, 26, 176] can potentially expose JIT bugs occasionally, they encounter severe challenges to prevent them from effectively exposing the JIT bugs. In particular, when *ClassFuzz* [28] randomly manipulates (e.g., deletes or inserts) instructions, it can also generate vast invalid/illegal seeds which essentially lead to testing

ineffectiveness. Although *Classming* [26] aims at improving over *ClassFuzz* for generating more valid seeds by intentionally breaching the variable dependencies to expose erroneous data flows, it potentially causes early-terminated JVM executions, i.e., insufficient testing of JITs. Meanwhile, noticing that *JavaTailor* [176] demands a preset database containing the Java programs executed to trigger JVM bugs, applying *JavaTailor* to specifically expose JIT bugs can be potentially challenging since the size of the JIT bug datasets are often limited and can only cover a small subset of possible bugs. To address these issues, it is essential for a fuzzer aiming at extending the usage of the JIT optimization techniques, which can be intuitively realized by proposing the fuzzing strategy to both advance the activation of the JIT optimization techniques and mutate control flows while preventing early termination of their associated testing runs.

Many fuzzers, e.g., AFL [171], MOPT [94], and Neuzz [131], adopt code coverage as guidance to facilitate bug/vulnerability exposure, i.e., generating mutants and retaining them as seeds for further mutations if they are executed to increase/optimize code coverage of target programs. However, code coverage can hardly be applied for general-purpose JVM testing techniques because JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [26]. On the other hand, we also notice that code coverage can be more deterministically captured for JITs. Therefore, it is plausible and potentially beneficial to adopt code coverage to guide our JIT testing.

6.2 Approach

We propose *JITfuzz*, a coverage-guided fuzzer for JVM JIT with two mutator types and a mutation scheduler. In particular, *JITfuzz* is implemented with Jimple-level instructions provided by *Soot* [147] which is a framework for analyzing and transforming JVM-based applications. Figure 6.1 presents the overall workflow of *JITfuzz*. Typically, Initialized with a *seed corpus*, *JITfuzz* iterates each *seed* to generate mutants under a given time budget. For each iteration, *JITfuzz* determines its mutation limit (i.e., the number of mutators applied to the given seed) according to the collected coverage updates. Note that in this chapter, *JITfuzz* develops four optimization-activating mutators and two control-flow-enriching mutators to facilitate the usage of JIT optimization techniques and enrich the control flows. Accordingly, *JITfuzz* adopts a mutator scheduler to select and schedule the mutators by the mutation limit based on a lightweight dynamic optimization algorithm to optimize the runtime code coverage. Eventually, if such resulting mutant increases code coverage in the target JIT, it is added to the *seed corpus* for further mutations.

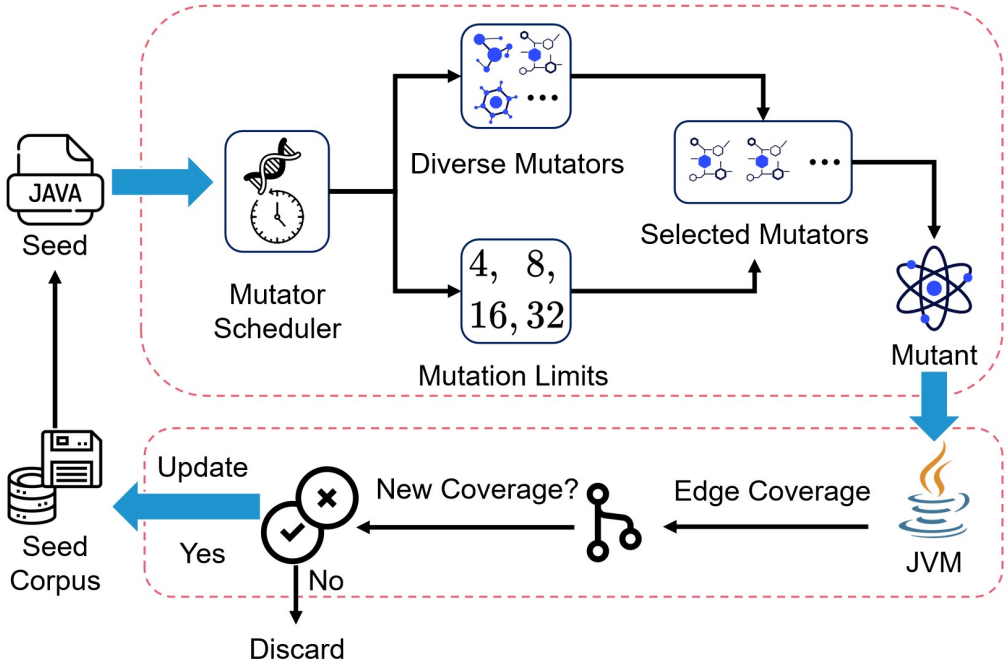


Figure 6.1: The framework of *JITfuzz*

6.2.1 Mutators

JIT adopts multiple optimization techniques to strengthen the runtime compilation performance of JVM-based programs. Many such optimization techniques involve complicated mechanisms, e.g., code analysis and semantics-preserving code transformations, potentially having the defects which can cause erroneous program executions. Intuitively, extensively applying such optimization techniques can advance JIT bug exposure. Accordingly, we determine to design optimization-activating mutators. Specifically, we first analyze typical JIT optimization techniques, e.g., simplification [137] and escape analysis [30, 54], and then derive the strategies which aim for extensively triggering the corresponding optimization techniques. As a result, such strategies are adopted as the optimization-activating mutators.

We further realize that mutating control flows in seeding class files for executing JIT can potentially advance the effectiveness of JIT fuzzing due to the following reasons. First, control-flow analysis can potentially advance the JIT optimization techniques by identifying where and how to optimize JVM-based programs [62]. Second, control-flow analysis can facilitate the correct compilation from Java programs into native machine code by verifying correctness of semantics [157], etc. At last, existing work [144, 163, 40, 26] also demonstrate that diversifying control flows in running programs can significantly increase fuzzing performance, e.g., coverage. Accordingly in *JITfuzz*, we propose a set of control-flow-enriching mutators including one statement-wrapping mutator to strengthen the usage of basic blocks and one transition-injecting mutator to strengthen the usage of their transitions respectively based on the control flow of a given seed. In

addition, they both are designed to preserve semantics correctness of target programs for preventing early-terminated testing runs, e.g., causing no verification errors.

Note that prior to proposing the mutators, one should adjust the setups of running JVMs such that JITs can be explicitly launched. To this end, we use the JVM command `java -Xcomp cls` for a given class `cls`.

Optimization-activating mutators

While optimization-activating mutators can be proposed in accordance with each existing optimization technique, exhaustively designing them can be cost-ineffective. In this chapter, we design four optimization-activating mutators corresponding to the representative optimization techniques in the existing JITs. i.e., function inlining [65], simplification [137], escape analysis [47], and scalar replacement [113] which are commonly adopted by the JITs from diverse well-recognized JVMs [107, 66, 42, 41]. Table 6.1 shows the details of the optimization-activating mutators via rewrite rules [13].

Table 6.1: Optimization-Activating mutators

Optimization techniques	Rules for Jimple-level optimization-activating mutators	Example

Function Inlining	`Rule 1:`
	$\|\|[e[\gamma = \alpha\ op\ \beta]\ end]\|\| \rightarrow$
	$\|\|[let\ function\ f(x,y) = x\ op\ y\ in\ e[\gamma = f(\alpha,\beta)]\ end]\|\|$
		`Example 1:`
		`-int i2 = i0 + i1;`
		`+public int inline(int i0, int i1) {`
		`+  return i0 + i1;`
		`+}`
		`+int i2 = inline(i0, i1);`

Simplification	`Rule 2:`
	$\|\|[e[\gamma = \alpha\ op\ \beta]\ end]\|\| \rightarrow$
	$\|\|[let\ expr = 0\ in\ e[\gamma = \alpha\ op\ \beta + expr]\ end]\|\|$
		`Example 2:`
		`-int i2 = i0 + i1;`
		`+int i3 = new Random().nextInt();`
		`+int i2 = (i0 + i3) + (i1 - i3);`

Scalar Replacement & Escape Analysis (ArgEscape level)	`Rule 3:`
	$\|\|[e[\gamma = \alpha\ op\ \beta]\ end]\|\| \rightarrow$
	$\|\|[let\ obj.field = \alpha\ in\ e[\gamma = obj.field\ op\ \beta]\ end]\|\|$
		`Example 3:`
		`-int i0 = 0;`
		`+Digit r0 = new Digit(0);`
		`+int i0 = r0.integer;`

Escape Analysis (GlobalEscape level)	`Rule 4:`
	$\|\|[e[\gamma = \alpha\ op\ \beta]\ end]\|\| \rightarrow$
	$\|\|[let\ this.field = \alpha\ in\ e[\gamma = this.field\ op\ \beta]\ end]\|\|$
		`Example 4:`
		`-Object i0 = new Object();`
		`+this.object = new Object();`
		`+Object i0 = this.object;`

Function-inlining-activating mutator. Noticing that function inlining refers to merging the instructions of small-scale functions into their callers to reduce the cost of function calls, our function-inlining-activating mutator is proposed to replace a randomly selected instruction with a function where only such instruction is contained, as Rule 1. More specifically, given an expression $\alpha\ op\ \beta$ (op denotes a binary operator), we create a new function $f(x,y)$ which returns the expression $x\ op\ y$. Consequently, we mutate the original expression $\gamma = \alpha\ op\ \beta$ as its corresponding transformation $\gamma = f(\alpha,\beta)$ in the given program context e. For instance, Example 1 shows that the original instruction `return i0 + i1` is mutated by a function `inline` containing it in order to facilitate the application of function inlining and test its capacity of merging small-scale functions into their callers.

Simplification-activating mutator. Noticing that simplification refers to simplifying an arithmetic expression, the simplification-activating mutator replaces a simplistic arithmetic expression as a semantics-preserving yet complicated expression. As in Rule

2, we update the original expression $\alpha\ op\ \beta$ with its semantics-preserving expression $\alpha\ op\ \beta + 0$ and generate an expression $expr$ calculated to be zero. Correspondingly, we mutate the original expression $\gamma = \alpha\ op\ \beta$ as $\gamma = \alpha\ op\ \beta + expr$. To illustrate, Example 2 shows that we mutate the instruction `i2 = i0 + i1` by adding and subtracting a randomly generated integer `i3` at the same time to facilitate the application of simplification on the expression.

Scalar-replacement-activating mutator. Scalar replacement essentially refers to investigating whether a stack variable can replace an object allocated in the heap in order to save memory resources. Accordingly, our scalar-replacement-activating mutator is proposed to replace stack variables with objects in the heap on target programs. Specifically, Rule 3 demonstrates that we first create an object *obj* and assign an existing variable α as its field, and then replace α with the field $obj.field$ in any original expression $\alpha\ op\ \beta$. Example 3 shows that we create a `Digit` object `r0` to mutate the constant integer `0` in the original instruction to be its associated value stored in `r0.integer` so as to facilitate the application of scalar replacement. Note that such mutator can also be used for the escape analysis in the *ArgEscape* level [30] when JIT verifies whether an object in the heap has side effects or not.

Escape-analysis-activating mutator. To facilitate the escape analysis in the *GlobalEscape* level [30], we also design an escape-analysis-activating mutator that replaces a local object α with the static field of the object $this.field$, which is referred by `this` pointer, as Rule 4. In Example 4, we first create the local object and assign it as the static field of `this` object, and then reassign the reference `i0` to `this.object`. Thus we create a *GlobalEscape* scenario to access the original local object `i0` via the static field `this.object` to facilitate the application of escape analysis.

Control-flow-enriching mutator

In this chapter, we propose a set of control-flow-enriching mutators to enrich the program control flows for augmenting the fuzzing effectiveness. Note that while the existing JVM fuzzers *Classming* and *ClassFuzz* also adopt control-flow mutators, applying them can easily cause early-terminated testing runs by generating random transitions and fail to expand the size of control flows, e.g., increase the number of basic blocks.

Statement-wrapping mutator. Intuitively, to increase the number of basic blocks in the existing control flows without devastating their executions, one can design *if(true)* statements and/or *loop(limit)* statements to contain the existing program statements. Accordingly, we design a statement-wrapping mutator to wrap a given statement within *if* and/or *loop* statement(s).

Algorithm 6 presents the details of applying a statement-wrapping mutator under an input instruction $\mathcal{I}$, its associated *seed*, and a counter *Cnt* denoting the runtime recursion depth. If *Cnt* exceeds threshold *LIMITATION*, the real-time resulting *mutant* is returned (lines 2 to 3). Otherwise, we randomly choose a design strategy to generate a new expression *expr* to contain $\mathcal{I}$ (line 4). One is to generate an *if(true)* block (lines 5 to

Algorithm 6: Statement-wrapping Mutator

```
1  Function DefaultControlFlow(I, seed, Cnt):
2      if Cnt ≥ LIMITATION then
3          return mutant ← seed;
4      strategy ← randomly select 0 or 1;
5      if strategy == 0 then
6          expr ← "if (true) {I;}";
7      if strategy == 1 then
8          expr ← "loop (limit) {I; limit -= 1;}";
9      mutant ← update with expr in the seed;
10     Cnt ← Cnt + 1;
11     return DefaultControlFlow(expr, mutant, Cnt);
```

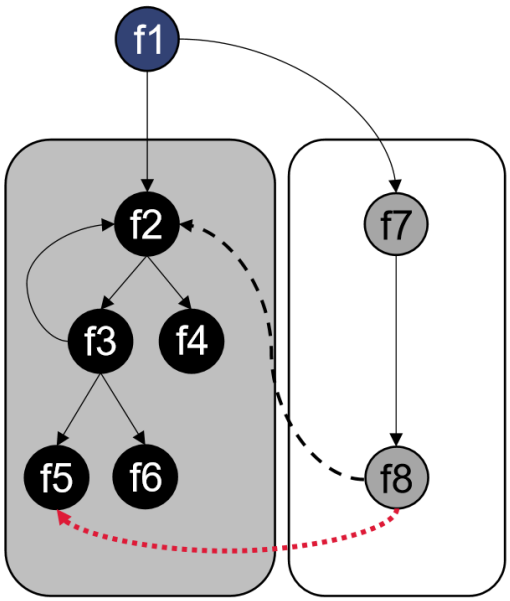


Figure 6.2: An example for illustrating the transition-injecting mutator

6). The other is to generate a *loop(limit)* block (lines 7 to 8). Then the *mutant* is derived by updating the *seed* with the resulting *expr* (line 9) followed by updating *Cnt* (line 10). Note that the above operations are recursively executed (line 11).

Transition-injecting mutator. We propose a transition-injecting mutator to enrich the transitions among basic blocks. While we simply select two random basic blocks for generating their transition, we also realize that without ensuring the correct execution of the associated target programs, it is likely to cause early-terminated program execution, i.e., the verification error caused by undefined variables. Figure 6.2 presents a real-world illustrative example (Since the code is complex, we demonstrate the complete code and its corresponding labels in [3] due to page limit) where the dark solid lines denote the existing transitions between basic blocks. Assume by applying the statement-wrapping mutators, a transition is established from `f8` to `f5` (denoted as the red dash line) and thus leads to an execution path `[f1, f7, f8, f5]`. However, since

`f5` depends on variables `sampleIndex` and `samplePos` [3] defined in `f3` instead of any of its predecessors on execution path `[f1, f7, f8, f5]`, this transition definitely causes an undefined variable error and prevents further testing on the deep states of JIT, e.g., the optimization techniques. Therefore, when designing the transition-injecting mutator, we also need to resolve the potential dependency issues so as to facilitate testing deep states of target programs. Note that according to *Soot* [147], any variables used by basic block β is defined either in β or its dominators. Therefore, to prevent undefined variable errors when creating a transition between two randomly chosen basic blocks (represented as $\alpha \rightarrow \beta$), it is essential to redirect the transition from α to a dominator of β that define all variables possibly used in β. However, there can be a possible side effect that if massive such transitions are redirected to the same dominator such as the identical entry basic block among similar seeds, their remaining partial control flows can be quite alike or even identical, causing limited program execution spaces and thus hindering program state exploration. Therefore in this chapter, for a randomly generated basic block transition $\alpha \rightarrow \beta$, we determine to redirect it from α to the closest dominator of β which causes no undefined variable errors when applying the transition-injecting mutator.

Algorithm 7 illustrates our transition redirection mechanism when applying the transition-injecting mutators. Given an input seed, we first construct its control-flow graph and identify its entry basic block (lines 2 to 3), and then generate a *directed CFG* by deleting all the transitions created by executing *loop* expressions or applying transition-injecting mutators (line 4) on top of the original CFG. After randomly selecting the source basic block `src` and the sink basic block `sink` (lines 5 to 6), we store all the dominators of `src` in order as the list `srcPre` (lines 9 to 11). Next, we also store all the dominators of `sink` in order as the list `sinkPre`. Accordingly, we identify the closest common dominator from `sinkPre` and `srcPre` as `sinkFiC` (lines 12 to 17). Finally, we identify the immediate post-dominator of `sinkFiC` in the set `sinkPre` as `sinkNew` and create a transition from `src` to `sinkNew` to generate a mutant (lines 18 to 19). For example in Figure 6.2, we first assume `f8` is `src` and `f5` is `sink`. Next, we obtain the closest common dominator for `f8` and `f5`, i.e., `f1`. Then, `f2` is identified as the immediate post-dominator of `f1` among all the dominators of `f5`. At last, we create transition `f8`→`f2` to generate a mutant.

We then discuss why Algorithm 7 can prevent early termination of program executions and limited program state exploration, i.e., the sink dominator `sinkNew` selected by Algorithm 7 is the closest dominator of `sink` causing no undefined variable error. First, assume that redirecting the transition from `src` to `sinkNew` incurs an undefined variable error in the original error-free program execution, i.e., a variable is used without a definition in `sinkNew` after redirection. Then we infer that this variable can be accessed by `sinkFiC` since `sinkFiC` dominates `sinkNew` in `sinkPre`. Accordingly, this undefined variable error can be spread in `sinkFiC` (otherwise, `sinkFiC` is obliged to define variables to prevent the undefined variable error in `sinkNew`), contradicting our

Algorithm 7: Transition Redirection Mechanism

```
1  Function RedirectBasicBlockTransition(seed):
2      controlFlowGraph ← obtainCFG(seed);
3      entry ← identifyEntry(seed);
4      directedCFG ← deleteSelectedEdges(controlFlowGraph);
5      src ← randomlySelectBasicBlock(directedCFG);
6      sink ← randomlySelectBasicBlock(directedCFG);
7      srcPre, sinkPre, sinkFiC ← {}, {}, {};
8      srcNxt, sinkNxt ← src, sink;
9      while entry ∉ srcPre do
10         srcNxt ← getDominator(srcNxt, directedCFG);
11         srcPre ← srcPre ∪ {srcNxt};
12     while entry ∉ sinkPre do
13         sinkFiC ← sinkPre ∩ srcPre;
14         if sinkFiC ≠ ∅ then
15             break;
16         sinkNxt ← getDominator(sinkNxt, directedCFG);
17         sinkPre ← sinkPre ∪ {sinkNxt};
18     sinkNew ← get the immediate post-dominator of sinkFiC from sinkPre;
19     mutant ← create a transition from src to sinkNew;
20     return mutant;
```

assumption. Thus, redirecting the transition to `sinkNew` is ensured to incur no undefined variable error. Next, we discuss why `sinkNew` is the closest dominator of `sink` which the transition can be redirected to without causing any undefined variable error. Similarly, assume there is a post-dominator of `sinkNew` which the transition can be redirected to without causing any undefined variable error. Since redirecting the transition from `src` to the post-dominator of `sinkNew` does not cause any undefined variable error, we infer that `sinkNew` is not allowed to define new variables (otherwise, it is possible that the variable is only defined in `sinkNew` which is not included in the execution after the transition redirection, causing an undefined variable error in its post-dominator). Such inference contradicts the fact that `sinkNew` is allowed to define new variables as a dominator of `sink`. Thus, `sinkNew` is ensured to be the closest dominator of `sink` causing no undefined variable error.

6.2.2 Mutator Scheduler

After proposing multiple mutator types to strengthen the usage of the optimization techniques in JITs in *JITfuzz*, how to aggregate their strengths to optimize their overall effectiveness becomes our next challenge. To this end, intuitively, an optimization guide is essential. Note that while JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [26] (such that coverage usually cannot be applied as a guide for testing JVMs), coverage updates can be more deterministically captured for JITs. Therefore, we determine to adopt the runtime coverage updates of target programs for guiding our mutator scheduling plans.

In this chapter, we build our mutator scheduler upon the UCB-1 algorithm [59], a lightweight algorithm which constructs an optimistic guess to the expected payoff of each action and picks the action with the optimal payoff to guide future iterative executions. The UCB-1 algorithm is adopted to schedule the mutators in *JITfuzz* due to the following reasons. First, in *JITfuzz*, scheduling mutators to optimize their aggregated effectiveness at runtime essentially is a stochastic optimization problem which exploits limited knowledge (i.e., runtime coverage). Notably, the UCB-1 algorithm is proposed to exactly address such stochastic optimization problem and has been widely adopted for similar tasks [59, 149, 120]. Next, scheduling mutators for fuzzing essentially demands limited overhead such that adequate computing resources can be leveraged for key technique components, e.g., mutations, program executions, and coverage collections. Notably, the UCB-1 algorithm yields rather limited overhead, i.e., quickly adjusting the mutator options based on runtime coverage updates, to approach the optimal solutions for each iterative execution. As a result, *JITfuzz* utilizes the UCB-1 algorithm to schedule the mutators according to runtime coverage updates. In particular, for a given seed, *JITfuzz* first identifies a mutation limit to determine how many mutators should be scheduled. Similar to AFL [171], *JITfuzz* adopts multiple mutation limit options (four in this chapter, i.e., 4, 8, 16, 32). Next, by the scheduled mutation limit, *JITfuzz* repeatedly selects one mutator out of all the six possible options (i.e., four optimization-activating mutators and two control-flow-enriching mutators). More specifically, the mutators can be scheduled via Equation 6.1 where $result(t)$ denotes both the mutation limit result and the corresponding mutator result at the t-th iteration.

$$result(t) = argmax_j(\frac{1}{t_j}\sum_{i=1}^{t_j} x_{ji} + \sqrt{\frac{2\ln(t-1)}{t_j}}) \tag{6.1}$$

In Equation 6.1, t_j denotes the total number that the j-th mutator/limit option has been selected till the t-th iteration, and x_{ji} refers to the reward of the j-th mutator/limit option in the i-th iteration. Accordingly, the selected mutators are applied to the given seed in turn for generating the mutants at the t-th iteration. Meanwhile, the obtained coverage is recorded to update the scheduler as a reward, which is one if the coverage is increased and zero otherwise.

6.2.3 Discussion

We consider the concept of *JITfuzz* can be inspiring for more fuzzing domains where multiple components are included in one testing target. In particular, when proposing mutants to cope with the target program as a whole instead of their individual components, it is possible that the generated mutators fail to fully access their components and thus compromise the fuzzing effectiveness. On the other hand, applying the mutators for addressing individual components only might still incur testing ineffectiveness, since they are highly likely to cause divergent testing effects respectively [94].

Therefore, it is promising to include an additional mutator scheduler to augment their collective power.

6.3 Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness of *JITfuzz* on a real-world benchmark suite composed of 16 projects. In particular, we first collect 6 projects commonly adopted from prior work [26, 176] and 10 additional popular open-source Java projects in *GitHub* [52]. Next, we compare *JITfuzz* with state-of-the-art mutation-based JVM fuzzer *Classming* [26] and generation-based JVM fuzzer *JavaTailor* [176] in terms of the edge coverage results obtained from JIT of our target JVM, and evaluate the effectiveness of different components of *JITfuzz*. In particular, we attempt to answer the following research questions:

- ***RQ1:*** *Is JITfuzz effective in fuzzing JIT?*
- ***RQ2:*** *Are the different components of JITfuzz effective in terms of ablation study*?

Moreover, we report and analyze the bugs on our adopted benchmark exposed by *JITfuzz*. Note that all the evaluation details are presented in our *GitHub* page [1].

6.3.1 Benchmark Construction

In this chapter, we define a set of rules to collect influential real-world Java projects in *GitHub* to form our benchmark for our evaluation. In particular, we search with the keyword "Java" on *GitHub*, and then randomly select 10 projects with decent star number (larger than 100) and LoC numbers (larger than 10k). In addition, we select all 6 projects from state-of-the-art *JavaTailor* which are all excerpted from the dacapo benchmark [9] with 102 stars. Table 6.2 demonstrates the detailed information of our benchmark suite where the top 6 projects are adopted by *JavaTailor*. Furthermore, for each of these projects, we randomly select one of the ten classes with the highest cyclomatic complexity [97] as the seed program where the cyclomatic complexity refers to the number of linearly independent paths through the source code of a class [6, 58, 16, 78].

6.3.2 Environment Setup and Implementation

We perform our evaluations on a work station, with AMD EPYC 7H12 CPU and 256 GB memory. The operating system is 64-bit Ubuntu 18.04.5 LTS. We choose HotSpot (Java 19) [107] as our target JVM to obtain the coverage results and set the *LIMITATION* for Algorithm 1 to two. Note that the results of more *LIMITATION* setups are presented in our *GitHub* link [1] due to page limit. Moreover, following prior work [171, 131, 159], we adopt the edge coverage obtained from JIT to reflect the effectiveness of our studied techniques. To collect runtime edge coverage, we first obtain all the source files of JIT only. Next, we utilize the partial instrumentation tool from AFL++ [49] to

Table 6.2: Benchmark information

Project	Stars	LoC	Initial class(seed)
avrora	102	111k	...avrora/Main.class
eclipse		26.4k	...EclipseStarter.class
pmd		197.4k	...pmd/PMD.class
jython		360.7k	...python/util/jython.class
fop		331.3k	...fop/cli/Main.class
sunflow		28.5k	...sunflow/Benchmark.class
hutool	23.5k	265.7k	..core.text.PasswdStrength.class
javapoet	9.7k	12.4k	...ClassName.class
mybatis-3	17.5k	161.3k	...ibatis.parsing.GenericTokenParser.class
zxing	29.9k	219.3k	...zxing.qrcode.encoder.Encoder.class
fastjson	24.8k	103.9k	...fastjson.JSON.class
guice	11.3k	110.6k	...inject.spi.InjectionPoint.class
commons-text	242	54.7k	...commons.text.numbers.ParsedDecimal.class
rocketmq	17.8k	178.2k	...rocketmq.filter.util.BloomFilter.class
spark	9.3k	23.1k	spark.resource.UriPath.class
vert.x	3.3k	215.4k	...vertx.core.json.JsonArray.class

instrument such source files for runtime edge collection. Note that JVM does have non-deterministic coverage such as garbage-collection mechanism [26]. However, since we only include the source files of JIT, such issues are mitigated while we collect the edge coverage information during fuzzing JIT.

All our experiments are run for 24 hours following prior work [159, 131, 130]. Note that all the experimental results are averaged from five runs to reduce the impact of randomness.

6.3.3 Result Analysis

RQ1: the effectiveness of *JITfuzz*

Table 6.3 demonstrates the evaluation results of *JITfuzz*, *JavaTailor* and *Classming* in terms of edge coverage.

Overall, we observe that *JITfuzz* can significantly outperform *Classming* in terms of the edge coverage. Specifically, *JITfuzz* can explore averagely 34,939 edges, while *Classming* only explores 27,306 edges, i.e., *JITfuzz* explores over 27.9% more edges than *Classming*. Meanwhile, we also find that *JITfuzz* outperforms *JavaTailor* by 18.6% (34,939 edges vs. 29,451 edges). Moreover, *JITfuzz* can significantly outperform *Classming* and *JavaTailor* on each individual project (from 4.9% to 1.8× for *Classming* and from 3.0% to 98.4% for *JavaTailor*). Such results altogether reflect that *JITfuzz* achieves significant edge coverage advantages over *Classming* and *JavaTailor*.

We also apply the Mann-Whitney U test to illustrate the significance of *JITfuzz* in Table 6.3. We can observe that the p-values of *JITfuzz* comparing with other studied

fuzzers in terms of average edge coverage are far below 0.05, which indicates *JITfuzz* outperforms all studied fuzzers significantly.

> *Finding 1: JITfuzz is more effective than Classming and JavaTailor by exploring 27.9% and 18.6% more edges on average.*

Table 6.3: Effectiveness of the *JITfuzz* mutators

Benchmark	*JITfuzz*	*Classming*	*JavaTailor*	Variants by Disabling Mutators						Other Variants	
				$JITfuzz_{-scalar}$	$JITfuzz_{-escape}$	$JITfuzz_{-simp}$	$JITfuzz_{-inline}$	$JITfuzz_{-wrap}$	$JITfuzz_{-trans}$	$JITfuzz_{randtr}$	$JITfuzz_{randsch}$
hutool	37,006	32,994	33,345	34,209	34,483	35,276	32,018	29,378	30,374	31,367	35,038
javapoet	36,178	33,948	35,138	33,675	30,990	34,296	34,135	32,381	32,070	32,465	33,599
mybatis	29,209	18,708	21,642	23,165	22,354	25,358	24,126	22,874	24,642	16,560	26,064
zxing	36,184	28,350	33,406	35,214	30,161	34,931	35,680	34,474	28,731	29,682	32,156
fastjson	36,222	26,121	26,956	30,114	34,367	34,963	33,715	33,269	28,711	26,341	32,874
guice	36,852	30,757	32,257	29,805	31,241	30,571	24,521	26,937	28,227	26,734	32,144
commons-text	36,863	35,147	34,058	35,075	36,075	36,881	34,503	32,958	34,039	33,129	33,039
rocketmq	32,808	29,043	30,910	26,977	28,457	24,097	26,241	26,012	24,060	24,782	30,422
spark	33,936	12,130	17,109	28,145	24,270	24,854	32,568	28,613	26,279	25,638	29,695
vert.x	34,056	24,162	25,598	28,056	23,806	31,199	22,785	27,322	25,302	26,986	32,238
avrora	36,835	29,339	32,634	28,879	28,208	36,275	32,539	28,344	24,598	28,150	33,073
eclipse	33,057	30,299	29,692	29,518	26,841	31,556	28,035	24,086	24,933	24,925	28,692
pmd	36,985	28,898	32,505	30,263	30,123	29,788	30,522	27,963	28,801	25,441	32,623
jython	35,284	25,827	29,060	29,443	31,155	30,630	29,892	32,110	25,226	28,139	33,102
fop	32,713	22,972	26,782	27,439	27,934	29,878	31,426	29,269	28,880	26,912	30,453
sunflow	34,846	28,202	30,133	33,632	26,595	27,584	30,553	30,851	25,713	27,317	32,002
average	**34,939**	**27,306**	**29,451**	**30,225**	**29,191**	**31,133**	**30,203**	**29,177**	**27,536**	**27,160**	**31,700**
p-value	**N/A**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**

RQ2: Effectiveness of each component

In this section, we conduct a set of ablation studies to evaluate the effectiveness of the key technical components in *JITfuzz*.

Effectiveness of the mutators. to perform ablation studies on the effectiveness of each mutator, we build the following six variant techniques of the original *JITfuzz* by disabling the corresponding mutators, i.e., $JITfuzz_{-inline}$, $JITfuzz_{-simp}$, $JITfuzz_{-scalar}$, and $JITfuzz_{-escape}$, $JITfuzz_{-wrap}$, and $JITfuzz_{-trans}$ which respectively disables the function-inlining-activating mutator, the simplification-activating mutator, the scalar-replacement-activating mutator, the escape-analysis-activating mutator, the statement-wrapping mutator, and the transition-injecting mutator. Accordingly, we can derive the effectiveness of a mutator by comparing the performance of its associated technique variant with the original *JITfuzz*.

Table 6.3 demonstrates the edge coverage results of *JITfuzz* and our studied technique variants. In general, we can observe that *JITfuzz* outperforms each variant averagely from 12.2% to 26.9% in terms of edge coverage, which is rather substantial. Moreover, we also find that all the variant can outperform *Classming* in terms of edge coverage from 0.8% to 14.0% respectively and three variants outperform *JavaTailor* from 2.6% to 5.7%, which delivers the fact that the framework of *JITfuzz* is robust even with partial mutators. Such results suggest that all mutators are effective and integrating them together optimizes the performance in exploring edges for JIT.

> *Finding 2: Each mutator of JITfuzz is effective and integrating them optimizes the performance of exploring edges.*

Effectiveness of the transition redirection mechanism. We further investigate the effectiveness of the transition redirection mechanism for applying the transition-injecting mutator. As in Algorithm 7, the transition-injecting mutator redirects a randomly generated transition from the source to a dominator of the sink to prevent using undefined variables. Accordingly, we build a technique variant *JITfuzz*$_{randtr}$, which simply creates transition directly for two randomly chosen basic blocks without applying any redirection while retaining all other proposed mutators in this chapter.

We can observe from Table 6.3 *JITfuzz* significantly outperforms *JITfuzz*$_{randtr}$ by 28.6% on average in terms of edge coverage, indicating that redirecting transitions is essential to facilitate the efficacy of the transition-injecting mutator.

> *Finding 3: Transition redirection mechanism is essential for the transition-injecting mutator in augmenting its edge coverage performance.*

Effectiveness of the mutator scheduler. To investigate the effectiveness of mutator scheduler, we build a technique variant *JITfuzz*$_{randsch}$, which randomly schedules the mutators and the mutation limit in each iterative execution instead of applying the coverage-guided mutator scheduler.

In general, we can observe from Table 6.3 that mutator scheduler is effective since *JITfuzz* outperforms *JITfuzz*$_{randsch}$ significantly by 10.2% (34,939 vs. 31,700 explored edges) on average in terms of edge coverage. More specifically, we can further observe that *JITfuzz* outperforms *JITfuzz*$_{randsch}$ consistently upon all the benchmark projects. Such results indicate that our adopted mutator scheduler is rather powerful in strengthening the effectiveness of JIT fuzzing.

> *Finding 4: Adopting mutator scheduler can significantly improve edge exploration for JITfuzz by scheduling mutators and its mutation limit.*

6.3.4 Bug Report and Analysis

JITfuzz is effective in exposing multiple real-world JIT/JVM bugs where a bug is defined as a defect within a specific JVM version in this chapter. In our paper, a bug is exposed if a seed 1) triggers any JVM crash, or 2) incurs different outputs among different JVMs (e.g., one JVM normally terminates while another reports an exception). After careful manual analysis, we then report the potential bugs as well as the corresponding seeds to the developers. Specifically, we apply the seeds generated by *JITfuzz* in our evaluation to run multiple JVMs, i.e., different versions of OpenJ9 [104], OpenJDK [107], and OracleJDK [109], for exposing their bugs. Note that we tend to include their recent versions since they are typically adopted by a large-scale collection of real-world projects, i.e., potentially having more impact than the older ones.

Table 6.4: Issues found by *JITfuzz*

JVMs	# Issues Reported		# Issues Confirmed		# Issues Fixed	
	JIT	Non-JIT	JIT	Non-JIT	JIT	Non-JIT
OracleJDK	1	0	1	0	0	0
OpenJDK	18	5	13	5	7	5
OpenJ9	4	8	4	4	0	4
TOTAL	23	13	18	9	7	9

Table 6.4 demonstrates the detailed information where we have successfully detected 36 JVM bugs. After reporting them to the corresponding JVM developers, 27 of them have been confirmed and 16 have been fixed. More specifically, 23 of them are JIT bugs, 18 have been confirmed, and 7 are fixed by the developers. Note that none of the bugs can be detected by *Classming* or *JavaTailor*. To illustrate, we observe that executing many seeds adopted by *Classming* fail to activate JIT optimization techniques with the abrupt termination. Meanwhile, the database adopted by *JavaTailor* which contains the Java programs exposing JVM bugs fails to contain sufficient JIT bugs for reference. We then introduce four typical bugs exposed by *JITfuzz* as follows.

JIT segmentation fault

We reported a vulnerability [68] on the C2 compiler [112]—a specific JIT compiler in HotSpot (OpenJDK), which affects several OpenJDK versions, including 7u351, 8, 11, 17.0.2, 18, and 19. It was assigned with a bug ID JDK-8283441 and has been fixed later. This vulnerability is exposed by running the original JUnit tests with a seed generated from `JSON.class`. In particular, the affected JVMs crashed due to a segment fault within `ciMethodBlocks::make_block_at(int)`, as in Figure 6.3.

```
ciBlock *ciMethodBlocks::make_block_at(int bci) {
 ciBlock *cb = block_containing(bci);
 if (cb == NULL ) {
  ciBlock *nb = new(_arena) ciBlock(_method, _num_blocks++, bci);
  _blocks->append(nb);
  // segmentation fault
  _bci_to_block[bci] = nb;
  return nb;
 } else if (cb->start_bci() == bci) {
  return cb;
 } else {
  return split_block_at(bci);
 }
}

```

Figure 6.3: One C2 segmentation fault bug in HotSpot

The developers located the issue to two methods in the seed whose bytecodes end abruptly with unreachable basic blocks, as shown in Figure 6.4. They found that

```
416: return
// Unreachable block
417: iinc    5, 1
420: iload   5
422: iconst_2
423: if_icmple 339
(end of bytecode)
```

(a) JSON.config() code snippet

```
623: areturn
// Unreachable block
624: iinc    20, 1
627: iload   20
629: iconst_2
630: if_icmple 336
(end of bytecode)
```

(b) JSON.parseObject() code snippet

Figure 6.4: Unreachable basic blocks in the generated class

HotSpot builds control flow graphs for unreachable basic blocks where JIT fails to validate. As a result, by compiling unreachable basic blocks, JIT accesses invalid memory, causing the segmentation fault. Eventually, they fixed this issue as follows:

> *"The new verifier checks by bytecodes falling off the end of the method, and the old verify does the same, but only for reachable code. So we need to be careful of falling off the end when compiling unreachable code verified by the old verifier."*

Dead loop assertion failure

JITfuzz discovered a HotSpot vulnerability [68] on JIT caused by an assertion failure, indicating that a dead loop was detected as in Figure 6.5. The OpenJDK versions 8, 11, 17, 18, 19 and 20 are affected by this vulnerability which has been reported to the developers and assigned with a bug ID JDK-8280126.

```
1  void PhaseGVN::dead_loop_check( Node *n ) {
2   if (n != NULL && !n->is_dead_loop_safe() && !n->is_CFG()) {
3    bool no_dead_loop = true;
4    ...
5    if (!no_dead_loop) n->dump(3);
6    // assertion failure
7    assert(no_dead_loop, "dead loop detected");
8   }
9  }
10
```

Figure 6.5: One dead loop assertion in HotSpot

The developers confirmed this bug and tried to analyze the corresponding buggy class file but failed by applying the tools provided by OpenJDK. They implemented multiple helper functions to analyze the control structure and inferred that HotSpot may miscalculate the control flow and consider certain nodes to be unreachable. As a result, such nodes take unexpected data as input and cause a dead loop, e.g., a data node in control flow graph references itself directly or indirectly. Eventually, they decided to defer this issue to JDK 20 due to its complexity with the following feedback:

> *"The difficulty with this bug is that we have many paths that get eliminated, finding the real source of the issue feels like searching for a needle in a haystack."*

JIT crash during optimization

We reported an OpenJ9 vulnerability [106] on JIT optimizer, which has been confirmed by the developers. When applying option `optlevel` at the `hot` level or higher, JIT failed to compile a seed generated from `ParsedDecimal.class` due to a segmentation error within `TR_OrderBlocks::peepHoleBranchBlock`, as shown in Figure 6.6.

```
void TR_OrderBlocks::peepHoleBranchBlock(TR::CFG *cfg, TR::Block *block,
    char *title)
 {
 ...
 // crash
 TR::Block *fallThroughBlock = fallThroughEntry->
 getNode()->getBlock();

```

Figure 6.6: Assertion failure during verification

The developers found that a node created by `generalLoopUnroller` is `NULL`. Furthermore, this issue can be bypassed by setting the JVM option `disableGLU` to disable the `general loop unroller`. Eventually, developers concluded that JIT miscalculated the control flow graph to incorrectly move certain nodes.

> *"However If I add tracing on this method, the crash goes away. I'll instrument the code and see if I can catch the issue in an early stage of the optimization."*

Other runtime vulnerabilities

In addition to JIT vulnerabilities, *JITfuzz* also reveals runtime defects. For example, we reported an OpenJ9 vulnerability [105] on the verification stage, which causes OpenJ9 to crash due to an assertion failure.

To locate this issue, the developers reproduced the crash in a debug build and found that the crash occurred when releasing the stackmap frame memory at /runtime/verbose/errormessagehelper.c, as shown in Figure 6.7.

```
releaseVerificationTypeBuffer(StackMapFrame* stackMapFrame,
    MethodContextInfo* methodInfo)
{
 if (NULL != stackMapFrame->entries) {
  PORT_ACCESS_FROM_PORT(methodInfo->portLib);
  // crash
  j9mem_free_memory(stackMapFrame->entries);
 }
}

```

Figure 6.7: Assertion failure during verification

The developers further found that OpenJ9 incorrectly built stackmaps for the class file due to its huge size. In particular, OpenJ9 cannot allocate the memory for its locals

and stack and the crash occurs when it tries to release the memory of the stackmap frame. The developers replied as follows.

> *"There is no element of 'locals' and 'stack' in the current stackmap frame in which case the code above didn't handle at this point."*

Eventually, the developers fixed this issue by adding additional checks to handle the case with empty locals and stack.

To summarize, by adopting the optimization-activating mutators, the statement-wrapping mutators and the mutator scheduler with easy-to-capture coverage information, *JITfuzz* can expose multiple types of JIT bugs which can be hardly explored by the existing approaches.

6.4 Limitations

While the mechanisms of *JITfuzz* can be leveraged for fuzzing optimization mechanisms in other compilers, its implementation using Soot is tailored for JVM. Consequently, broader adoption of *JITfuzz* across diverse contexts necessitates significant engineering efforts. In forthcoming endeavors, we intend to extend the applicability of *JITfuzz* to encompass fuzzing scenarios involving compilers for various programming languages, such as Rust and Go.

6.5 Summary

In this chapter, we develop a coverage-guided fuzzing framework for JVM JITs, namely *JITfuzz*, which includes four optimization-activating mutators and two control-flow-enriching mutators. Moreover, *JITfuzz* adopts a lightweight mutator scheduler to schedule the mutation limit and the associated mutators for maximizing the overall effectiveness of fuzzing. To evaluate the effectiveness of *JITfuzz*, we construct a benchmark suite with 16 real-world JVM-based projects. Our evaluation results suggest that *JITfuzz* can outperform state-of-the-art *Classming* and *JavaTailor* by 27.9% and 18.6% in terms of the edge coverage on average. Meanwhile, we also demonstrate that all proposed mutators and the mutator scheduler are effective. Furthermore, *JITfuzz* successfully detects 36 previously unknown bugs none of which can be detected by *Classming* or *JavaTailor*. Specifically, 27 of them have been confirmed and 16 have been fixed by the corresponding developers. More specifically, 23 of them are JIT bugs, 18 have been confirmed, and 7 have been fixed.

Chapter 7

SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing

Testing JVMs via manually designing tests based on analyzing JVM semantics can be extremely challenging due to their intricacies, i.e., it is hard to generate sufficient high-quality inputs based on complicated JVM semantic rules to thoroughly test the program states of JVM executions. To address such a challenge, prior research work attempts to integrate fuzzing and differential testing [48] for automated JVM testing, i.e., designing fuzzers to generate class files as tests for executing different JVMs such that their discrepant execution results (defined as *inter-JVM discrepancies* in this chapter) can be used for testing analytics. For instance, *ClassFuzz* [28] fuzzes Java class files by mutating their modifiers or variable types to test the loading, linking, and initialization phases in JVMs. More recently, *Classming* [26] fuzzes live bytecode to mutate the control flows in class files to test deeper JVM execution phases (e.g., bytecode verifiers and execution engines) across multiple JVMs.

However, the power of the existing JVM fuzzers may not be fully leveraged since they fail to apply seed scheduling and mutator scheduling mechanisms which have become vital in enhancing fuzzing effectiveness. In particular, seed scheduling refers to aggressively selecting and mutating seeds to facilitate program bug/vulnerability exposure. Many coverage-guided fuzzers [171, 11, 131, 130, 94, 88] schedule seeds for mutation simply when executing them can increase code coverage. The existing JVM fuzzers, on the contrary, fail to leverage code coverage as seed scheduling guidance because they can hardly exploit the runtime coverage information for fuzzing since JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [28]. Specifically, *ClassFuzz* only collects coverage information for initializing JVMs and *Classming* even exploits no coverage for fuzzing. Similarly, while scheduling mutators guided by code coverage has been proven effective recently [94, 159], the existing JVM fuzzers are restrained by selecting mutators uniformly under no guidance.

In this chapter, we present *SJFuzz* (***S***cheduling for ***J***VM ***Fuzz***ing, in our *GitHub* repository [95]), a JVM fuzzing framework which applies seed and mutator scheduling

mechanisms to facilitate the exposure of discrepant execution results among different JVMs, i.e., inter-JVM discrepancies, for JVM differential testing. Specifically, *SJFuzz* schedules seeding class files under two types of guidance—discrepancy and diversity. On one hand, *SJFuzz* retains the class files that can be executed to directly incur inter-JVM discrepancies or used to generate mutants for sufficiently testing JVMs, i.e., avoiding early termination on JVM testing process, as seeds for further mutations. On the other hand, assuming that increasing code coverage can be reflected by diversifying test case (class file) generation, *SJFuzz* applies a coevolutionary algorithm [32] to filter the remaining class files to augment class file diversity for further mutations. Moreover, *SJFuzz* also iteratively schedules mutators to augment the overall distances between seed and mutant class files. In particular, for a given seeding class file, *SJFuzz* estimates the diversity expectation of each mutator and selects a mutator to optimize the class file diversity.

To evaluate *SJFuzz*, we conduct a set of experiments upon various popular real-world JVMs, e.g., OpenJDK, OpenJ9, DragonWell, and OracleJDK. In particular, we apply *SJFuzz* and *Classming*, the state-of-the-art mutation-based JVM fuzzer, to generate class files via seeding class files selected from popular open-source Java projects, which are then executed in the studied JVMs to expose their discrepancies. Moreover, to further demonstrate the power of *SJFuzz*, we also include *JavaTailor* [176], a generation-based JVM fuzzer that utilizes existing JVM historical bug-revealing test programs to expose JVM discrepancies, as our baseline. The results suggest that *SJFuzz* significantly outperforms *Classming* in terms of inter-JVM discrepancy exposure, e.g., exposing 3.5×/6.3× more total/unique discrepancies on average. Meanwhile, *SJFuzz* also outperforms the *JavaTailor* by 5.2%/14.3% in terms of total/unique inter-JVM discrepancy exposure. Moreover, we have reported 46 potential issues to their corresponding developers after analyzing the inter-JVM discrepancies incurred by *SJFuzz*. As of submission time, 20 bugs have already been confirmed by the developers.

7.1 Motivating Example

In this section, we introduce a real-world JVM bug exposed by applying differential testing via mutating program control flows to illustrate the potential issues of state-of-the-art *Classming* and motivate *SJFuzz*. Specifically, Figure 7.1 shows a simplified Jimple code snippet of a mutated method `findResources()` in `AntClassLoader.class` from project `Ant`, where the Jimple code representation refers to a Soot-based intermediate representation of Java programs for simplifying Java bytecode analysis [146]. Running such a class file exposes an execution discrepancy between OpenJDK (1.8.0_232) and OpenJ9 (1.8.0_232). Specifically, in the original seeding class file, after `z0` is assigned with the value of member `ignoreBase` of `r1`, i.e., 0 (line 11), line 12 is immediately executed, followed by line 19. However, inserting the `lookupSwitch` instruction changes the control flow to be from line 8 to line 13. Next, method `getResources()` of

```
1  protected Enumeration<URL> findResources(...){
2  +  i0 = 5
3     ...
4     r4 = r1.parent
5     ...
6  +  i0 = i0 + -1
7  +  if i0 <= 0 goto line 9
8  +  lookupswitch(i0) { case 4: goto line 4; default: goto line 13; }
9     r3 = $r5
10    ...
11    $z0 = r1.ignoreBase // r1.ignoreBase is always 0
12    if $z0 == 0 goto line 19
13    $r7 = specialinvoke r1.getRootLoader()
14    if $r7 != null goto line 16
15    ...
16    $r8 = specialinvoke r1.getRootLoader()
17    $r9 = virtualinvoke $r8.getResources(r2)
18    ...
19    $r6 = staticinvoke CollectionUtils.append(r3, r14)
20    return $r6
21 }
```

Figure 7.1: A class file generated under diversification guide.

the root class loader is invoked (line 17). As a result, by passing parameter META-INF/`MANIFEST.MF` to `getResources()`, OpenJDK8 (1.8.0_232) returns nothing while OpenJ9 (1.8.0_232) returns the paths of the `MANIFEST.MF` files in its lib JARs. We further found such a discrepancy was triggered as OpenJDK8 failed to find the existing resources.

Although this discrepancy can be exposed by simply inserting a `lookupSwitch` instruction to the seeding class file, *Classming* failed to expose such a discrepancy under multiple runs in practice. Specifically, we found that for the example class file, *Classming* updated new seeding class files after iteratively generating mutants via uniformly selected mutators, and failed to reproduce the discrepancy-inducing execution path under a fair time limit. This fact suggests that adopting similar seeding class files via only uniformly selected mutators may hinder the effective exploration of discrepancy-inducing mutants. Furthermore, we also observe that while *Classming* typically adopted similar seeding class files mutated from one root throughout causing inefficient usage of computing resources, which can be leveraged to explore other promising seeding class files, e.g., the ones that can expose multiple discrepancies [101, 102] simultaneously. This fact also leads to a demand of scheduling multiple seeding class files other than only one in one run based on their discrepancy-guided potentials.

To conclude, inspired by previous works [**pathschedule**, 11, 94, 24], all these insights motivate our proposed approach *SJFuzz*, which effectively schedules seeding class files and diversifies their mutations to increase the chances of exposing discrepancies for JVM testing.

7.2 The approach of *SJFuzz*

The framework of *SJFuzz* is demonstrated in Figure 7.2. Overall, *SJFuzz* enables iterative mutation-based class file generation. In particular, given a seeding class file,

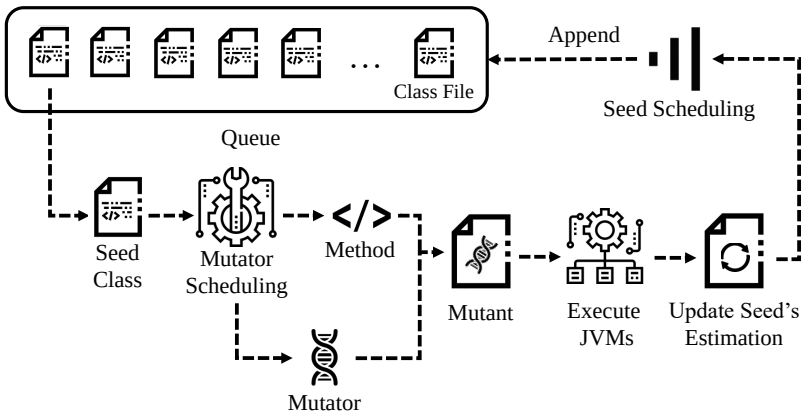


Figure 7.2: The framework of *SJFuzz*.

SJFuzz adopts the control flow mutation strategy to generate its mutant class file (Section 7.2.1). Accordingly, for each iteration, *SJFuzz* schedules seeding class files (Section 7.2.2) under the diversity and discrepancy guidance. *SJFuzz* also schedules mutators deterministically or randomly to augment class file diversity (Section 7.2.3) for further iterations.

Algorithm 8 shows the details of *SJFuzz*, which is initialized by adding one `seedClass` into the `queue` and assigning the `seedClass` to be *optional* . Under each iterative execution (line 6), *SJFuzz* schedules control-flow mutators for each class file in the `queue` to facilitate class file diversity (lines 7 to 8). Note that a newly generated class file is initialized as an *optional* seed (line 9). Such a seed and its parent (i.e., `mutantClass` and `class`) can be identified whether to be *primary* after running on the adopted JVMs (lines 10 to 14). For any valid mutant class file, *SJFuzz* updates its distance to its seeding class file to guide further mutator scheduling (lines 16 to 17), and the distance for each invalid mutant class file is updated to -1 (line 19). At last, all the *primary* class files and filtered *optional* class files are retained for future mutations (as the output of the discrepancy guidance and the diversity guidance, lines 20 to 22). After each iteration, the updated seeding class files are used for JVM differential testing. Such iterations are terminated when hitting the `budget`. Note that *SJFuzz* only enables valid class files for mutations because mutating an invalid class file tends to cause exceptional program behaviors rather than unexplored inter-JVM discrepancies.

7.2.1 Control Flow Mutation

Prior research work on fuzzing compilers including JVMs tend to mutate program control flows via a set of corresponding mutators for exposing bugs in their "deep" execution stages [40, 20, 83, 26]. Following such prior works (and also for a fair comparison with them), *SJFuzz* also adopts such control flow mutation with representative mutators. Specifically in source code level, *SJFuzz* randomly selects two original instructions,

Algorithm 8: The framework of *SJFuzz*

```
1  Function SJFUZZ_FRAMEWORK(seedClass, budget, bound):
2      queue ← list();
3      queue.add(seedClass);
4      setClassToOptional(seedClass);
5      while total budget has not exceeded do
6          for class in queue do
7              method ← randomlySelectMethod();
8              mutantClass ← scheduleMutator(class, method);
9              setClassToOptional(mutantClass);
10             runJVMs(mutantClass);
11             if mutantClass incurs new DISCREPANCIES then
12                 setClassToPrimary(class);
13                 if mutantClass is VALID then
14                     setClassToPrimary(mutantClass);
15             if mutantClass is VALID then
16                 distance ← Levenshtein(class, mutantClass);
17                 updateMutatorDistance(class, distance);
18             else
19                 updateMutatorDistance(class, -1);
20         primary ← retainPrimarySeeds(queue, bound);
21         option ← scheduleOptionalSeeds(queue, bound);
22         queue ← merge(primary, option);
23     return queue;
```

and creates a directed transition between them. If such a transition is a loop, the corresponding iteration will be limited to 5 times. Correspondingly, *SJFuzz* implements the mutators with the Jimple-level instructions `goto`, `lookupswitch`, and `return` provided in *Soot* [147].

SJFuzz iteratively selects random positions from randomly selected methods to apply the control-flow mutators. Specifically, *SJFuzz* establishes an instruction list that contains the instructions executed by the adopted JVMs under their execution order. Next, *SJFuzz* selects and inserts a control flow mutator into a random spot of the instruction list under each iteration.

7.2.2 Seed Scheduling

Since it is difficult to directly apply coverage guidance for fuzzing JVMs, we adopt two alternative types of guidance for our seed scheduler. In particular, we develop a discrepancy-guided seed scheduler which retains discrepancy-inducing class files for further mutations. We also develop a diversity-guided seed scheduler which filters other class files to augment the overall class file diversity via a coevolutionary algorithm [32].

Discrepancy-guided seed scheduling. Intuitively, if running a mutant of a class

file can cause inter-JVM discrepancies, such a class file is likely to generate more discrepancy-inducing mutants than other class files as it implies a potential connection with program bugs [101, 102]. Therefore, such a class file and its mutant (if valid, i.e., successfully running in at least one JVM under test without unexpected behaviors such as verifier errors or crashes) are defined as *primary* class files and are retained for future iterative executions. This seed scheduler is essentially similar to many coverage-guided seed schedulers [171, 11, 131, 130, 88], which tend to retain seeds when running them can increase code coverage.

Diversity-guided seed scheduling. When running a mutant of a class file does not instantly cause inter-JVM discrepancies, it does not necessarily suggest that no discrepancy-inducing mutants can be generated in future iterative executions. In other words, leveraging such class files can also possibly advance the inter-JVM discrepancy exposure. In this chapter, such class files are characterized as *optional*. Note that differential testing usually enables vast space for generating test cases, which indicates that the total number of class files that cause discrepancies between sophisticated JVMs can be rather limited. We can then infer that the *optional* class files may significantly outnumber the primary class files. Therefore, *SJFuzz* should filter the *optional* class files to ensure fuzzing efficiency.

We consider that increasing code coverage can be essentially reflected as diversifying execution paths on the same JVMs and thus demands diverse test cases (i.e., class files). Therefore, our seed scheduler for *optional* class files is guided by class file diversity, so that the *optional* class files are filtered to augment class file diversity. In particular, how to measure class file diversity should be resolved in the first place. To accurately reflect the fine-grained differences between class files, an ideal metric is expected to reflect their instruction-by-instruction comparisons. Therefore, we adopt the executed instruction list of a given class file, namely *EntryInstruction*, as the representative instructions to efficiently measure the diversity between JVM class files. Note that *EntryInstruction* reflects the instruction-level execution order and retains only the unique executed instructions to reduce the ambiguity of diversity measurement caused by repeated instructions, e.g., in loops. Eventually, we measure the diversity between a pair of class files by deriving differences between their associated *EntryInstruction*s.

In this chapter, *SJFuzz* applies edit distance, i.e., Levenshtein Distance [87], a metric widely used to derive the "minimum number of single-character edits (insertions, deletions, or substitutions)" between two strings, to measure the difference between *EntryInstruction*s because the mutation-based class file generation can analogize the single-character string edits as demonstrated in [23]. To be specific, *SJFuzz* generates class files by mutating the selected seeding class files, i.e., inserting the instructions with the adopted control flow mutators. The resulting iterative single-point mutations between the seed and mutant class files can be modeled as inputs for Levenshtein-Distance-based computation when such class files are all modeled as "strings". For instance, assume two *EntryInstruction*s of their corresponding class files $C_1 : [i_1, i_2, i_3, i_4, ..., i_n]$

and $C_2 : [i_1, i_3, i_4, ..., i_n]$. We can observe that C_1 can be transformed from C_2 by only inserting one instruction i_2 between i_1 and i_3. Therefore, their Levenshtein Distance is computed as 1.

Accordingly, *SJFuzz* adopts a coevolutionary algorithm [32] to efficiently evaluate the individual *optional* class files out of their group by constructing its fitness function to reflect their average distances with other *optional* class files. Specifically, for each *optional* class file, *SJFuzz* calculates its total Levenshtein Distance with other *optional* class files. Subsequently, the average Levenshtein Distance is calculated as the fitness score of the given class file. By sorting all the derived fitness scores, *SJFuzz* retains the top-N corresponding *optional* class files for further mutation-based class file generation, where N is predefined as the `bound` variable in Algorithm 8.

To illustrate, we incorporate the discrepancy- and diversity-guided seed schedulers to facilitate code coverage and inter-JVM discrepancies when differentially testing JVMs.

7.2.3 Mutator Scheduling

Since it is computationally expensive to derive the exact diversity of the overall class files on the fly, *SJFuzz* schedules mutators to diversify the seed and mutant class files under each iteration to approximate the overall class file diversity instead. In particular, *SJFuzz* first applies the edit distance, i.e., Levenshtein Distance [87] in this chapter, to delineate the diversity between a pair of class files. Accordingly, *SJFuzz* establishes a *deterministic mutator scheduling mechanism* for estimating the mutator that can optimize the seed-mutant distance. Meanwhile, *SJFuzz* also develops a *random mutator scheduling mechanism* to prevent the potential local optimization that can derive local optimal mutators caused by the *deterministic mutator scheduling mechanism*. As a result, *SJFuzz* derives a mutator for a given class/method by combining the two mechanisms.

Deterministic mutator scheduling. Note that any mutator selected from one iteration can incur cumulative impact on the mutations of the subsequent iterations. To capture such cumulative impact from the previous mutations, *SJFuzz* adopts the *Monte Carlo method* [100] to develop the *deterministic mutator scheduling mechanism*, where given a selected method m_p, *SJFuzz* develops a value function, represented as $V(c_i, a_j, m_p)$, to determine the mutation opportunity of class file c_i by applying a mutator a_j as demonstrated in Equation 7.1. Such a value function can reflect the resulting diversity of the overall class files under the mutation, i.e., the cumulative diversity expectation between the seed and mutant class files under all the iterations.

$$V(c_i, a_j, m_p) = \mathbb{E}[\frac{1}{N}\sum_{k=0}^{N} Distance_k(c_i, a_j, m_p)] \tag{7.1}$$

Here *Distance* refers to the Levenshtein Distance between the seeding class file c_i and its mutant class file by applying mutator a_j upon method m_p. $Distance_k(c_i, a_j, m_p)$

Algorithm 9: Mutator Scheduling

```
1 Function Fmainclass, method, explorationRate:
2     rand ← Random();
3     if rand < explorationRate then
4         mutator ← selectRandomMutator(class, method);
5         mutatedClass ← mutate(mutator, class);
6     else
7         bestMutator ← selectDeterministicMutator(method);
8         mutatedClass ← mutate(bestMutator, class);
9     return mutatedClass;
```

refers to their Levenshtein Distance in the k_{th} iteration which can be dynamically updated since the mutation spot is randomly selected in m_p under each iteration. $\mathbb{E}$ refers to the mathematical expectation of *Distance*s. It can be derived that $V(c_i, a_j, m_p)$ for c_i is incrementally updated and inefficient to be directly computed. Therefore, we further enable dynamic updates on $V_k(c_i, a_j, m_p)$ as presented in Equation 7.2 to approximate its value, where α is a constant. Note when one class file c_i fails to generate a valid class, its *Distance* is set to -1.

$$V_k(c_i, a_j, m_p) = V_{k-1}(c_i, a_j, m_p) + \alpha(Distance - V_{k-1}(c_i, a_j, m_p)) \tag{7.2}$$

As a result, we select a mutator a_j corresponding to the largest $V(c_i, a_j, m_p)$ for c_i. Typically, *SJFuzz* allows computing $V(c_i, a_j, m_p)$ after running the mutant class files on JVMs such that it can be used for mutator selection of the subsequent iteration when needed (as in line 17 of Algorithm 8).

Random mutator scheduling. Only maximizing $V(c_i, a_j, m_p)$ tends to cause local optimization, i.e., $V(c_i, a_j, m_p)$ is likely to converge to one mutator after iteratively selecting it, while the actual optimal mutator cannot be derived until later iterations. To address such an issue, *SJFuzz* further leverages a *random mutator scheduling mechanism* to reduce its possibility to select a sub-optimal mutator under early-terminated executions by randomly selecting one mutator for class file generation for the ongoing iteration. As a result, by properly combining such a mechanism with the *deterministic mutator scheduling mechanism*, it can potentially extend the *Monte Carlo* process until convergence for enhancing the selection probability of the optimal mutator, i.e., preventing the local optimization.

The overall mutator scheduling mechanism is presented in Algorithm 9. We first set an `explorationRate` and generate a random value for comparison (line 2). Next, if such a random value is less than the `explorationRate`, *SJFuzz* chooses the *random mutator scheduling mechanism* to return a random mutator for the ongoing iteration (lines 3 to 5). Otherwise, *SJFuzz* derives the mutator via the *deterministic mutator scheduling mechanism* (lines 6 to 8).

7.3 Evaluation

We conduct a set of experiments on various popular JVMs. Note that we include two state-of-the-art JVM fuzzers—the mutation-based fuzzer *Classming* [26] and the generation-based fuzzer *JavaTailor* [176] in our evaluation for performance comparison. In particular, *Classming* fuzzes live bytecode to mutate the control flows in class files as mentioned, and *JavaTailor* generates class files from JVM historical bug-revealing test programs (provided by the authors). Overall, we aim to compare *SJFuzz* with them in terms of their resulting inter-JVM discrepancies, the class file generation efficiency, and the reported bugs by answering the following research questions:

- **RQ1:** Is *SJFuzz* effective in exposing inter-JVM discrepancies?
- **RQ2:** Are the seed and mutator schedulers effective?
- **RQ3:** Is the diversity guidance effective?

Moreover, we report and analyze the bugs detected by *SJFuzz* with all the evaluation details presented in our *GitHub* page [95].

7.3.1 Benchmark Construction

We adopt multiple widely-used real-world JVMs, i.e., OpenJDK, OpenJ9, DragonWell, and OracleJDK, for running *SJFuzz* to expose their execution discrepancies. Note that their detailed versions are available on our *GitHub* page [95] since multiple versions of each JVM are used in our evaluation. We also adopt state-of-the-art mutation-based approach *Classming* and generation-based approach *JavaTailor* as the baselines for comparison as they outperformed other existing JVM differential testing approaches [26, 176]. Specifically, for a fair comparison with *SJFuzz* which integrates differential testing and test generation, we also run *Classming* and *JavaTailor* on all studied JVMs in parallel.

To launch *SJFuzz*, we adopt 26 class files via randomly sampling from 7 well-established open-source projects as the seeding class files for mutation-based class file generation. To construct such benchmarks, we first attempt to collect all available class files originally adopted for evaluating *Classming*, for approaching a fair performance comparison. As a result, `Eclipse`, `Jython`, `Fop`, and `Sunflow` are selected due to their availability while others incur stale configurations, JAR incompatibility, mismatched `main` declarations, etc. Moreover, we also adopt `Ant` and `Ivy` (two popular command-line applications from Apache Projects [5]) and `JUnit` [72] (a widely used unit testing framework) to expand our benchmark diversity.

Note that while the existing approaches, e.g., *Classming* and *ClassFuzz*, are designed to only launch mutations for the entry methods corresponding to the `main` methods, in this chapter, we attempt to adopt diverse "entry" modes, i.e., diverse method types (entries) for mutation. Particularly, we adopt two such modes: *main-entry* and *JUnit-entry*. More specifically, in addition to *main-entry* adopted by [28, 27], the new

JUnit-entry mode, on the other hand, refers to mutating other entry methods associated with JUnit tests of the seeding class files. To our best knowledge, we are the first to execute the unit tests of the seeds in compiler testing.

JUnit-entry can benefit the class file generation for the following reasons. First, *JUnit-entry* supplements *main-entry* on the mutation space for a class file which cannot be explored by *main-entry* only, since a large amount of JUnit test classes are designed for non-main methods in practice. Next, the execution discrepancies between JVMs are likely to be better presented in *JUnit-entry*, since assertions examine different JVM executions upon class files and thus enable smaller scope in exposing discrepancies and easier analytics than *main-entry*.

In this chapter, for *main-entry*, we select seeding class files as the class files containing the `main` methods. For *JUnit-entry*, since each project contains various test classes, we randomly adopt 4 class files under test for each project with more than 5 corresponding test methods from the projects `Fop`, `Jython`, `Ant`, `Ivy`, and `JUnit` which all use the JUnit framework with available test source files on the corresponding *GitHub* repositories. Note that `Fop`, `Jython`, `Ant`, and `Ivy` are chosen as both the *main-entry* and *JUnit-entry* benchmarks for straightforward performance comparison between the two modes within one project.

7.3.2 Environmental Setups

We perform our evaluation on a desktop machine, with Intel(R) Xeon(R) CPU E5-4610 and 320 GB memory. The operating system is Ubuntu 16.04. The `exploreRate` for Algorithm 9 is set to 0.1 and the `bound` for Algorithm 8 is set to 20 by default.

Similar as prior work [28, 10, 131, 11, 130], all benchmarks are executed by all the studied approaches for 24 hours to generate class files to reflect a large enough testing `budget`. Note that we run each experiment 20 times [76] for obtaining the average results to reduce the impact of randomness.

7.3.3 Result Analysis

RQ1: the Inter-JVM Discrepancy Exposure Effectiveness of *SJFuzz*

Note that in this chapter, to identify unique discrepancies, we first summarize the symptoms of the discrepant JVM behaviors, including assertions in JUnit tests, exceptions, and the results printed in standard output. Then we compare such symptoms with the previously recorded unique discrepancies to distinguish whether they are unique or not. More specifically, we first divide the overall output results into two categories—*non-exception output* (e.g., assertions in JUnit tests and results printed in standard output) and *exception*. Typically, an exception can be respectively presented for different JVMs. We then represent an exception as a tuple of its type and location from the output result. Furthermore, given one exception on all tested JVMs, one

Table 7.1: Discrepancies exposed by *SJFuzz*, *Classming* and *JavaTailor*.

Project	Benchmark(.class)	*SJFuzz*		*Classming*		*JavaTailor*	
		Total	Unique	Total	Unique	Total	Unique
Eclipse	EclipseStarter	2776.4	9.8	248.2	1.0	2487.2	8.3
Fop	Fop	22.3	1.9	1.0	1.0	21.1	1.4
Jython	Jython	800.5	7.0	14.1	1.0	874.3	7.4
Sunflow	Benchmark	1764.9	8.7	348.8	1.0	1574.2	6.0
Ant	Launcher	2514.6	11.8	486.1	1.0	2587.9	12.8
Ivy	Main	5042.9	19.1	1557.9	2.8	4679.9	14.9
Fop (JUnit)	FopConfParser	1612.1	9.9	84.3	0.8	1417.2	7.8
	FopFactoryBuilder	1651.2	9.3	382.9	3.0	1844.2	9.7
	ResourceResolverFactory	214.2	3.0	22.2	1.0	203.9	2.5
	FontFileReader	796.3	5.8	111.9	1.0	791.4	5.3
Jython (JUnit)	PyByteArray	2142.7	10.1	932.4	1.9	2209.2	10.3
	PyFloat	928.7	4.2	260.9	1.0	897.1	3.1
	PySystemState	232.4	3.8	56.5	1.0	222.3	3.0
	PyTuple	1661.8	7.2	312.6	1.1	1434.9	5.0
Ant (JUnit)	AntClassLoader	298.1	10.7	6.2	1.0	290.9	9.8
	DirectoryScanner	71.5	8.9	1.8	1.0	67.0	7.8
	Project	53.1	1.8	0.0	0.0	49.9	1.4
	Locator	1573.7	8.9	447.1	1.0	1485.4	7.8
Ivy (JUnit)	ResolveReport	211.7	4.7	0.0	0.0	197.7	4.0
	ApacheURLLister	645.9	7.2	213.4	1.0	605.2	6.2
	Configurator	509.0	7.1	57.5	1.0	470.9	6.0
	IvyEventFilter	1218.6	8.6	386.2	2.0	1109.7	7.7
JUnit (JUnit)	RuleChain	972.3	14.3	264.9	1.0	763.3	9.5
	TestWatcher	817.9	2.2	228.6	1.0	786.6	1.6
	ErrorReportingRunner	1986.6	10.6	506.9	1.0	1872.2	9.3
	Money	914.3	11.2	88.3	1.0	914.4	13.0
Average		1208.0	8.0	270.0	1.1	1148.4	7.0
p-value		N/A		2.35e-84		0.039	

discrepancy is formed by collecting, analyzing, and combining all their exception information. If no such a discrepancy was collected before, it is considered unique. The implementation code for this process can be found at [36].

The inter-JVM discrepancy results (both the total and the unique discrepancies) after executing the generated class files are presented in Table 7.1. For instance, for benchmark `Jython.class`, *SJFuzz* exposed a total of 800.5 discrepancies and 7.0 unique discrepancies averagely. We can observe that overall, *SJFuzz* can significantly outperform *Classming* in terms of the inter-JVM discrepancy exposure. To be specific, *SJFuzz* can expose 1208.0 total discrepancies and 8.0 unique discrepancies on average, while *Classming* can expose 270.0 total discrepancies and 1.1 unique discrepancies on average, i.e., *SJFuzz* exposes over 3.5×/6.3× more total/unique discrepancies than *Classming*. Moreover, we can further find that for all the adopted benchmark projects, *SJFuzz* can significantly outperform *Classming* in terms of both the total and unique discrepancy exposure. Note that *SJFuzz* also exposes all the discrepancies found by *Classming* in our evaluation. Meanwhile, we can observe that *SJFuzz* can also outperform *JavaTailor* by 5.2% more total discrepancies (1208.0 vs. 1148.4) and 14.3% more unique discrepancies (8.0 vs. 7.0) respectively.

Furthermore, we apply the Mann-Whitney U test to illustrate the significance of *SJFuzz*. It can be seen in Table 7.1 that the p-value of *SJFuzz* comparing with *Classming* in terms of the average unique discrepancies is far below 0.05 in each benchmark, which indicates that *SJFuzz* outperforms *Classming* significantly ($p < 0.05$). We can also observe that the p-value of *SJFuzz* comparing with *JavaTailor* in terms of the average unique discrepancies is also below 0.05 (0.039). Such results can reflect that *SJFuzz* can be quite effective.

Interestingly, we can observe that the advantage of *SJFuzz* over *JavaTailor* is not quite obvious as over *Classming*, i.e., *JavaTailor* is a more powerful baseline. We infer it is mainly because *JavaTailor* adopts a database containing a variety of historical JVM-bug-revealing test programs which can be quite enlightening for testing tasks. Nevertheless, as a typical data-driven approach, it can be naturally prone to common issues, e.g., data dependency and extra effort on maintaining the database. Surprisingly, *SJFuzz*, a lightweight end-to-end approach, can still outperform *JavaTailor* in exposing both total and unique discrepancies, indicating the power of our adopted mechanism of seed and mutator scheduling.

Finding 1: SJFuzz is effective by exposing 6.3× more unique discrepancies averagely than Classming (8.0 vs. 1.1), and 14.3% more unique discrepancies averagely than JavaTailor (8.0 vs. 7.0) under the same evaluation setups.

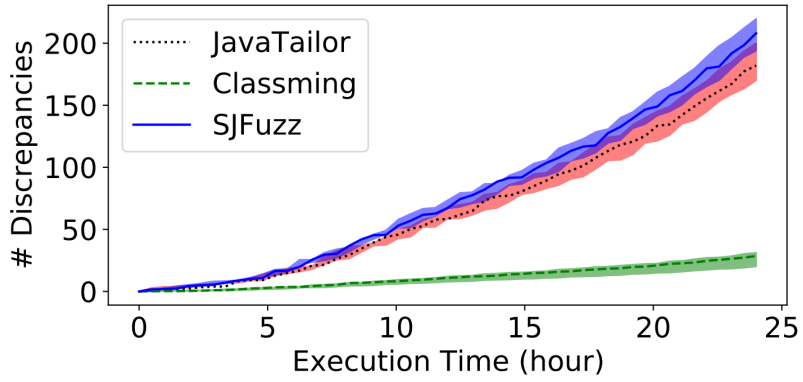
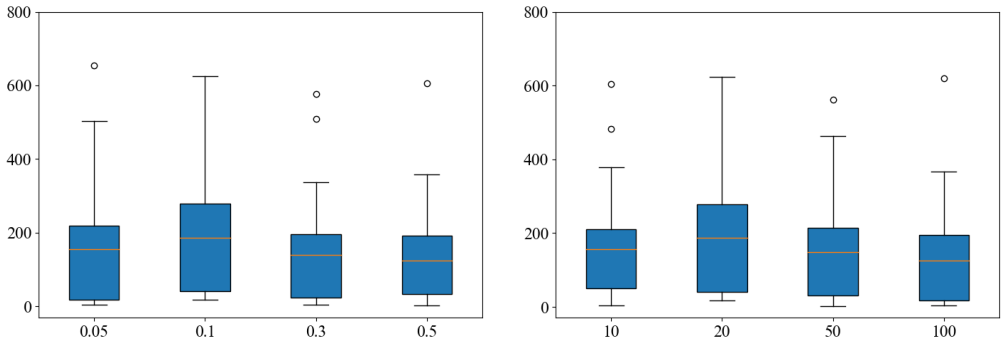


Figure 7.3: *SJFuzz*/*Classming*/*JavaTailor* efficiency in 24 hours.

We further investigate the impact of the execution time on discrepancy exposure by *SJFuzz*, *Classming*, and *JavaTailor*. Figure 7.3 shows how the exposed unique discrepancies on all the benchmarks by the three approaches vary over time. We can observe that although we enhanced the differential testing efficiency by running JVMs in parallel for *Classming* and *JavaTailor* (as in Section 7.3.1), we can also observe that *SJFuzz* can in general consistently outperforms *Classming* and *JavaTailor* in finding JVM discrepancies all the time before terminating the executions. Such results can further indicate the power of the discrepancy guidance mechanism of *SJFuzz*.

We also investigate the discrepancies exposed by our adopted entry modes for class file mutations: *main-entry* and *JUnit-entry*. Specifically, *SJFuzz* can significantly outperform *Classming* under both the entry modes, i.e., *SJFuzz* can expose 58.3 unique discrepancies in 6 *main-entry* benchmarks and 149.5 unique discrepancies in 20 *JUnit-entry* benchmarks, while *Classming* can only expose 7.8 and 21.6 unique discrepancies under such two entry modes respectively. *JavaTailor* also exposes 50.8 unique discrepancies in 6 *main-entry* benchmarks and 130.8 unique discrepancies in 20 *JUnit-entry*



(a) Exploration Rate. **(b)** Bound.

Figure 7.4: The impact of the parameter settings on *SJFuzz* in all benchmarks.

benchmarks. Additionally, for the projects which enable both *main-entry* and *JUnit-entry* (i.e., `Fop`, `Jython`, `Ant`, and `Ivy`), *SJFuzz* exposes 111.2 unique discrepancies in total under *JUnit-entry* and 39.8 under *main-entry*. Such a result indicates the effectiveness of our newly proposed *JUnit-entry* mode for JVM testing. We highly encourage future researchers/practitioners to look into the *JUnit-entry* mode for advancing JVM testing.

> *Finding 2: JUnit-entry is more effective than main-entry in exposing inter-JVM discrepancies.*

We have also observed that *SJFuzz* has rather stable performance across different configurations. To evaluate the impact of the parameter settings on *SJFuzz*, we evaluate the unique discrepancies in terms of different `explorationRate` (in Algorithm 9) and `bound` (in Algorithm 8) values on our benchmark suite, as presented in Figure 7.4. In particular, we set `bound` to the default 20 and investigate the impact of different `explorationRate`, i.e., 0.05, 0.1, 0.3 and 0.5 (Figure 7.4(a)). We also set `explorationRate` to the default 0.1 and investigate the impact of different `bound` (Figure 7.4(b)). Each box plot presents the distribution of exposed unique discrepancies for one configuration across all our studied subjects. We can observe that different configurations exert limited impact on the performance, indicating the effectiveness and stability of *SJFuzz*.

RQ2: Effectiveness of the Seed and Mutator Schedulers

In this section, we investigate the effectiveness of the seed and mutator schedulers respectively.

Effectiveness of the seed scheduler. To investigate the effectiveness of the adopted seed scheduler of *SJFuzz*, we record the number of discrepancies exposed by the *primary* and *optional* class files of the original *SJFuzz* approach, denoted as *SJFuzz*(primary) and *SJFuzz*(optional), respectively. Furthermore, we also build the two variant techniques

Table 7.2: Average number of discrepancies found by the studied techniques upon all benchmark projects.

Studied Subjects	All Discrepancies	Unique Discrepancies
SJFuzz(primary)	381.3	2.9
SJFuzz(optional)	826.7	5.1
$SJFuzz_{pg}$	252.3	1.2
$SJFuzz_{ef}$	764.7	4.8
JavaTailor	1148.4	7.0
$SJFuzz_{uniform}$	819.6	5.4
Classming	270.0	1.1
SJFuzz	1208.0	8.0

of *SJFuzz*: (1) $SJFuzz_{pg}$, which only activates discrepancy-guided seed scheduling for *SJFuzz*, and (2) $SJFuzz_{ef}$, which equally filters the class files regardless whether they are *primary* or *optional*. Note that $SJFuzz_{pg}$ retains the initial class file for further mutations until it explores a *primary* class file given that the initial class file is not *primary*.

In general, we can observe from Table 7.2 that $SJFuzz_{pg}$ can be effective by exposing 252.3 total discrepancies and 1.2 unique discrepancies on average. Interestingly, only $SJFuzz_{pg}$ itself can enable quite close performance with *Classming* (270.0 total discrepancies and 1.1 unique discrepancies on average as in Table 7.1). Such results can indicate the effectiveness of our "discrepancy-guided" intuition, i.e., exploiting the power of discrepancy-inducing class files can advance JVM differential testing.

Interestingly, Table 7.2 demonstrates that by integrating $SJFuzz_{pg}$ and $SJFuzz_{ef}$, i.e., applying the original *SJFuzz*, mutating *primary* class files can incur significantly more inter-JVM discrepancies, i.e., 381.3 vs. 252.3 total discrepancies with 2.9 vs. 1.2 unique discrepancies between *SJFuzz*(primary) and $SJFuzz_{pg}$. Such results indicate that injecting *optional* class files for test case generation can advance the *primary* class files to generate more discrepancy-inducing class files. To illustrate, when mutating *optional* class files generate discrepancy-inducing mutant class files, they are all converted to be *primary*. Thus, *primary* class files are increasingly adopted for further mutations such that their chances to expose discrepancies can be augmented. Furthermore, the fact that *SJFuzz*(optional) outperforms $SJFuzz_{ef}$ suggests that directly retaining *primary* class files for further mutations can also advance the *optional* class files to expose inter-JVM discrepancies. We can infer that by independently mutating *primary* class files via revoking their filtering process, more *optional* class files can be retained for further mutations because the *primary* class files no longer compete against them for being selected. To summarize, $SJFuzz_{pg}$ and $SJFuzz_{ef}$ can mutually advance each other to optimize the performance of *SJFuzz*.

Finding 3: As different components of the seed scheduling mechanism, $SJFuzz_{pg}$ and $SJFuzz_{ef}$ are both effective and integrating them can further advance each other in terms of exposing inter-JVM discrepancies.

Effectiveness of the mutator scheduler. To investigate the effectiveness of the mutator scheduler of *SJFuzz*, we build a variant technique $SJFuzz_{uniform}$ of the original *SJFuzz* by selecting mutators uniformly. Overall, we can observe from Table 7.2 that $SJFuzz_{uniform}$ can expose 819.6 total discrepancies and 5.4 unique discrepancies averagely. Although $SJFuzz_{uniform}$ still outperforms *Classming* 2.0×/3.9× averagely in exposing total/unique discrepancies, the exposed discrepancies decrease significantly after disabling mutator scheduling, i.e., 32.2%/32.5% averagely in exposing total/unique discrepancies compared to *SJFuzz*. Moreover, *JavaTailor* can outperform this variant by 40.1%/29.6% averagely in exposing total/unique discrepancies. Such results indicate that applying the mutator scheduler in *SJFuzz* can make significant contributions in terms of inter-JVM discrepancies.

> *Finding 4: Applying the mutator scheduling mechanism can significantly improve the power of exposing discrepancies for JVM fuzzers.*

RQ3: Effectiveness of the Diversity Guidance

The previous findings of the effectiveness of different *SJFuzz* components can imply their underlying mechanism of diversifying class file generation can be potentially effective. However, accurately measuring data diversity can be rather challenging. In this chapter, we delineate class file diversity in terms of the average seed-mutant Levenshtein Distance of the collected class files. Note that since *JavaTailor* is a generation-based approach (i.e., generating new class files with a variety of JVM-bug-revealing test programs can naturally result in significantly large Levenshtein Distances), we thus only include *Classming* in this discussion.

The diversity results of the class file generation are presented in Figure 7.5. We can observe that *SJFuzz* incurs much larger average seed-mutant Levenshtein Distance compared with *Classming*, i.e., overall 23.7× larger and 32.3×/22.5× larger under *main-entry*/*JUnit-entry*. It can be inferred that *Classming* tends to generate similar mutants, which also indicates the effectiveness of *SJFuzz*'s diversity-guided class file generation mechanism.

We further attempt to infer the possible reasons behind the diversity performance difference between *SJFuzz* and *Classming* in terms of the seed-mutant Levenshtein Distance. Assume a mutated class file (simplified version) in Figure 7.6 with only one executed instruction (line 4). Initially, *Classming* would select and insert the `return` mutator (line 5) because other mutators can result in the potential def-use violation of the `r0`-exclusive variables and thus the verification error. Such an error can hinder the detection of "deep" bugs, e.g., bugs incurred in execution engine. However, under this circumstance, the class file in Figure 7.6 is likely to be retained as the seed to repeatedly select the `return` mutator under each iterative execution for further class file generation. As a result, all the mutant class files realize single-mutation difference with their respective seeds, i.e., leading to potential short seed-mutant Levenshtein Distance. In

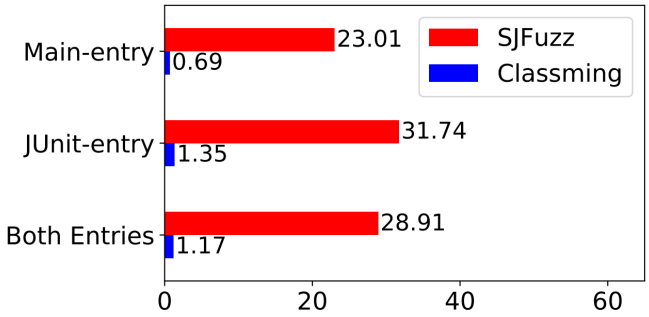


Figure 7.5: Average seed-mutant Levenshtein Distance.

```
class A {
  ...
  public void someFunction() {
    r0=<java.lang.System: java.io.PrintStream out>;
    return; // inserted by return mutator
    ......
  }
}
```

Figure 7.6: An example of mutating paradox for *Classming*.

contrast, *SJFuzz* is free from such constraints because it can diversify seed *optional* class files with best effort. Moreover, even when a seed *optional* class file generates a similar mutant class file, they together are hardly retained for further mutations under the diversity-guided class file filtering mechanism.

> *Finding 5: Diversifying class file generation is advanced in testing the "deep" bugs in execution engine by retaining sufficient valid class files.*

At last, we study the effectiveness of our adopted distance metric, i.e., the Levenshtein Distance, which is used to measure class file diversity. To this end, we adopt more distance metrics [8, 67, 51] for discussing their impact on *SJFuzz*. Specifically, we adopted Gestalt Pattern Matching Distance [51], Bag Distance [8], and Jaro Distance [67] in our evaluation. In particular, Gestalt Pattern Matching Distance adopts the number of matching characters and the length of strings to measure the similarity for two given strings. Bag Distance utilizes the maximum length between the relative complements [33] of two given strings' character sets with respect to each other to measure their distance. Jaro Distance is also a type of edit distance to calculate the similarity of two strings. As shown in Figure 7.7, distance metrics exert limited impact on the effectiveness of *SJFuzz* in terms of exposing unique discrepancies. Interestingly, we can observe that adopting edit distances, i.e., Jaro Distance and Levenshtein Distance, can achieve slightly better performance than other distance metrics.

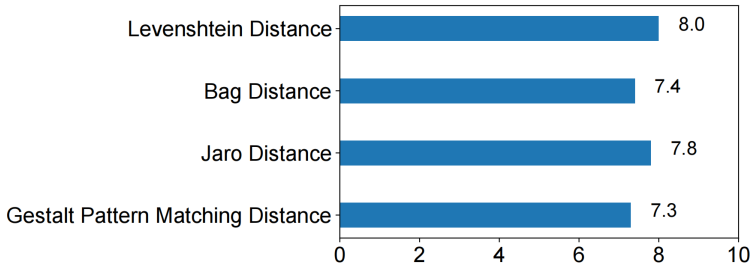


Figure 7.7: Average unique discrepancies exposed by different distance metrics in all benchmarks.

Table 7.3: Issues found by *SJFuzz*.

JVMs	# Reported				# Confirmed			
	Loading Phase	Linking Phase	Run-time	Crash	Loading Phase	Linking Phase	Run-time	Crash
OracleJDK	0	0	3	0	0	0	2	0
OpenJDK	2	3	3	3	0	0	2	0
Dragonwell	1	2	2	0	0	0	0	0
OpenJ9	6	7	12	2	0	4	10	2
TOTAL	9	12	20	5	0	4	14	2

7.3.4 Bug Report and Discussion

We manually analyze all the collected discrepancies to derive potential issues. Note that in this chapter, we define a bug as an error or an unexpected behavior for a specific JVM version. As a result, we report 46 potential issues from the discrepancies found by *SJFuzz* (*Classming* fails to expose any of these issues while *JavaTailor* can expose 25 of them), as in Table 7.3, to their corresponding developers. As of today, 20 were confirmed while 16 were fixed by the developers. The remaining 4 confirmed bugs are marked as "won't fix". We present some example bug reports as follows.

Resource Retrieval Bug

We reported an OpenJDK bug on retrieving JAR information which was confirmed by the OpenJDK developers with a bug ID `JDK-8244083`. This bug was exposed by the execution discrepancy between OpenJ9 and OpenJDK. Specifically, they both executed one class file from `AntClassLoader.class`, where OpenJDK failed to retrieve the JAR information from given resources while OpenJ9 succeeded. The developers inferred that certain side effect changed the behaviors of the original method.

Runtime Inconsistency Bug

We have reported an OpenJ9 bug on issuing a runtime erroneous return under the mutated classes from `Money.class`, as shown in Figure 7.8. We applied its original JUnit tests on all the class files mutated from `Money.class` which resulted in multiple

```
class A {
  ...
  public boolean isZero() {
    int var1 = this.amount();
    // OpenJ9 and OpenJDK get var1 = 0 here
    boolean var2;
    if (var1 == 0) {
      var2 = true; // OpenJDK executed here
    } else {
      var2 = false; // OpenJ9 executed here
    }
    return var2;
  }
}
```

Figure 7.8: Runtime inconsistency bug in OpenJ9.

errors/discrepancies. In particular, OpenJ9 reported an `AssertionError` while OpenJDK passed the test. However, when we further removed one JUnit test which caused `StackOverflowError`, both OpenJ9 and OpenJDK passed the test. Accordingly, we summarized that such a discrepancy may be caused by the unresolved dependency between JUnit tests and reported it to the corresponding developers [70].

To tackle such an issue, developers applied option `optlevel` at the `warm` level and inferred this as a JIT issue. After checking the `tree simplification` (an optimization feature in OpenJ9), developers found that OpenJ9 made a wrong assumption to the `nodeIsNonZero` flag set. As a result, the instruction `ificmpne` was changed to `goto` by OpenJ9 and it caused the associated branch to be always executed, even when the value of the associated variable did not meet the branch conditions. Eventually, they fixed this issue as follows:

> *...the `nodeIsNonZero` flag was set because IL gen assumed that slot 0 was still being used to store the receiver and thus the flag did not need to be reset. There is a method that is supposed to check if slot 0 was re-used so that flags can be reset. This problem can be fixed by adding cases to handle other types of stores to slot 0... I will open a pull request to make this change.*

Verifier Bug

A verifier bug usually is derived by analyzing the discrepancies about throwing a `verifyError` or not. In particular, verifier bugs are perceived typical "deep" bugs, i.e., bugs that are tricky to be detected and debugged.

By executing the mutated `ErrorReportingRunner.class` from project `JUnit`, we discovered that OpenJDK (1.8.0_232), OpenJDK (9.0.4), and OpenJDK (11.0.5) threw `VerifyError`, while OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) wrongly took it as a valid class file for execution. Moreover, there even incurred a discrepancy among multiple OpenJ9 versions, i.e., OpenJ9 (9.0.4) threw a `VerifyError`. Accordingly, we inferred that OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) were buggy and reported them to developers.

```
1 if (!verifyData->createdStackMap) { // enable to fix another issue
2   if (liveStack->uninitializedThis
3   && !targetStack->uninitializedThis) {
4     rc = BCV_FAIL;
5     goto _finished;
6   }
7 }
```

Figure 7.9: OpenJ9 buggy code in rtverify.c.

Interestingly, it took the developers quite a while to understand the cause of such bugs. At first, they speculated this issue as an "out of sync" problem:

> *It seems the code in verifier is likely out of sync or some new changes related to verifier were only merged for OpenJDK8 & OpenJDK11 given that only OpenJDK9/OpenJ9 captured VerifyError. Need to further analyze to see what changes in verifier caused the issue.*

When they attempted to locate the issue by checking the exception table, they found no exception table for the associated method of the mutated class file. Next, they divided the issue into two different checking branches: one was investigating the `simulateStacks` for how it propagated the `uninitalizedThis` (a variable to mark the status of `simulateStacks`) which may or may not be launched in the `mergeStacks` code; the other was comparing the differences in `rtverify.c` for different JVM releases. Finally, by comparing different versions of `rtverify.c`, the developers have identified that a checking mechanism on `uninitializedThis` was disabled in `matchStack()` when creating the `stackmap`. Accordingly, OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) were confirmed to fail to capture the `VerifyError`.

The buggy instructions of `rtverify.c` are demonstrated in Figure 7.9. OpenJ9 (1.8.0_232)/OpenJ9 (11.0.5) were allowed to correctly throw `VerifyError` when enabling the checking mechanism on `uninitializedThis` by removing line 1 in Figure 7.9. However, since such a checking mechanism was designed to prevent a Spring verifier issue [138], it could not be removed simply. Meanwhile, even though the `VerifyError` could be correctly captured on OpenJ9 (9.0.4), its associated `VerifyError` message was rather out-dated. Such issues together deliver a potential demand on upgrading the verification logic of OpenJ9. At last, the developers have stated that they intend to generate a patch to fix all of the exposed issues [148].

Controversy-Arousing Issue

In addition to assisting developers in exploring the "deep" bugs, we even triggered an in-depth discussion and revisit to the validity of well-established JVM mechanisms via a potential issue reported by *SJFuzz*.

We have found an issue that OpenJ9 could break the structured locking. In particular, when executing the corresponding mutated class `DirectoryScanner.class`,

```
    ...
66   return0 // return without exitmonitor
    ...
265   monitorenter // enter the monitor
    ...
709   invokestatic 911 Print.logPrint
712   iload 5
714   iconstm1
715   iadd
716   istore 5
718   iload 5
720   ifle 66 // go to line 66
    ...
```

Figure 7.10: The IllegalMonitorStateException issue of OpenJ9.

OpenJDK threw an `IllegalMonitorStateException` because the executing thread accessed a method and executed `entermonitor`, but simply returned without executing `exitmonitor`. However, OpenJ9 did not throw `IllegalMonitorStateException`. Accordingly, we inferred that OpenJ9 allowed returning a method under mismatching `entermonitor` and `exitmonitor` (which broke structured locking) and have reported it to the OpenJ9 developers [141]. Figure 7.10 refers to the partial class file that exposed this issue.

At first, the developers denied the potential violation of structured locking, i.e., they analyzed our submitted class file and claimed no violation of structured locking. However, during our further investigation, we discovered that while `exitmonitor` has not been executed from line 265 to line 720 in Figure 7.10, line 720 was executed followed by a `return` instruction (line 66) where `IllegalMonitorStateException` should have been thrown. Correspondingly, the developers reconsidered this issue and finally agreed on the violation of structured locking.

Since the developers still insisted on the legitimacy of their development schemes, they further questioned and inspected the validity of the structured lock mechanism.

> *We may end up with cleaner locking code if we enforced structured locking. This also came up recently in a discussion on how to handle OSR points for inlined synchronized methods. We should investigate the benefits/costs of adopting Structured Locking.*

By tracing back to the JVM specification [145] on the structured locking mechanism, the developers argued that structured locking could be allowed, yet not required. As a result, they considered revoking structured locking to be more as an domain-specific adaptation, rather than a bug, controversially.

In summary, *SJFuzz* is capable of detecting multiple types of "deep" bugs via exposing inter-JVM discrepancies for testing analytics. Furthermore, the bugs detected by *SJFuzz* can be rather tricky to be explored by the existing approaches, e.g., the bug incurred by unresolved JUnit test dependency and the bug that urged developers to trace back to JVM specifications.

7.4 Limitations

In black-box testing scenarios such as CPU or chip testing, when coverage guidance proves ineffective, the core mechanism of *SJFuzz* can synergize with techniques like differential testing or alternative testing oracles to unveil bugs. Similarly, *SJFuzz*'s reliance on Soot implementation tailored for JVM fuzzing necessitates substantial additional design and engineering endeavors for its adaptation to alternative testing contexts. In the future, we endeavor to extend the principles underlying *SJFuzz* to embrace a broader spectrum of analogous testing domains.

7.5 Summary

In this chapter, we proposed *SJFuzz*, the first fuzzing framework using seed and mutator scheduling for automated JVM differential testing. Specifically, *SJFuzz* employs a discrepancy-guided seed scheduler which retains discrepancy-inducing class files and class files that generate discrepancy-inducing mutants. It also employs a diversity-guided seed scheduler which filters other class files via a coevolutionary mechanism to augment class file diversity for further mutations. Moreover, *SJFuzz* applies a mutator scheduler based on the *Monte Carlo method* to diversify the class file generation. To evaluate the efficacy of *SJFuzz*, we performed an extensive study to compare *SJFuzz* with *Classming*, the state-of-the-art mutation-based JVM fuzzer, on various real-world benchmarks. The results show that overall, *SJFuzz* significantly outperforms *Classming* in terms of exposing inter-JVM discrepancies for JVM differential testing, e.g., *SJFuzz* exposes 8.0 unique discrepancies while *Classming* only exposes 1.1 unique discrepancies averagely on all the studied benchmarks. We also compare *SJFuzz* with the generation-based approach *JavaTailor* in terms of exposing JVM discrepancies. The results also suggest *SJFuzz* outperforms *JavaTailor*, e.g., *SJFuzz* exposes 14.3% more unique discrepancies than *JavaTailor*. To date, we have reported 46 previously unknown potential issues discovered by *SJFuzz* to the JVM developers where 20 were confirmed as bugs and 16 were fixed.

Chapter 8

Related Work

8.1 Fuzzing

Among all the coverage-guided fuzzers, AFL [171] is a widely-used baseline by retaining the mutants which can be executed to increase code coverage as seeds for further iterative executions. Many fuzzers are implemented upon AFL. Li et al. [88] proposed Steelix to explore new coverage efficiently by observing more runtime states. Lemieux et al. [86] introduced the concept of rare branches and facilitated the fuzzing efficacy by focusing on rare branches. In order to improve the fuzzing effectiveness, researchers also attempt to integrate dynamic analysis techniques such as taint analysis with fuzzing, e.g., AFL++ [49]. Rawat et al. [121] proposed VUzzer to identify the input format of the target program via taint analysis, for avoiding early termination in fuzzing. Liang et al. [90] proposed PATA, a more advanced taint analysis technique that can identify the loop variables efficiently during fuzzing. Du et al. [43] proposed WindRanger, which leverages the power of deviation basic blocks to facilitate directed greybox fuzzing. Furthermore, many researchers also propose seed scheduling techniques for improving fuzzing effectiveness. Böhme et al. [11] proposed AFLFast to schedule seeds during fuzzing via a Markov chain model to improve the performance of AFL. She et al. [132] introduced K-scheduler, which schedules seeds according to the reachable edges and potential coverage gain. Zhang et al. [173] utilized path constraint as the guidance function to schedule the seeds for harvesting new edges. Zhang et al. [172] proposed MobFuzz, which models fuzzing as a multi-objective problem via a multi-armed bandit and then schedules the seeds based on a particular optimization goal derived from the chosen objective combination. Meanwhile, Chen et al. [24] proposed MEUZZ to schedule the seeds in hybrid fuzzing based on the knowledge learned from past seed scheduling decisions made on the same or similar programs. Researchers also adopt constraint solvers to explore deep program states. Cadar et al. [15] proposed the fundamental symbolic execution engine Klee for aiding the fuzzers in solving the program constraints during fuzzing via symbolic execution. Accordingly, Yun et al. [167] introduced QSYM to combine a concolic executor for solving complicated program constraints in a selected coverage-guided fuzzer to leverage the power of symbolic execution in fuzzing. Kukucka et al. [77] proposed CONFETTI to combine taint analysis

and concolic execution to fuzz Java programs. To solve the constraints more efficiently, Chen et al. [19] proposed JIGSAW to evaluate the generated seeds with constraints on a native function produced by Just-in-time compilation. Instead of adopting the SMT-solver as other constraint-solving-based fuzzers, Chen et al. [21] proposed Angora to solve program constraints by a gradient descent algorithm. In addition, Fuzzing is utilized to detect vulnerabilities in specific domains. Shen et al. [134] proposed Drifuzz to fuzz WiFi and Ethernet drivers with concolic executor. Garbelini et al. [50] proposed BrakTooth to fuzz arbitrary Bluetooth Classic (BT) devices via constructing a protocol state machine. Shou et al. [135] proposed Corbfuzz to fuzz the security policies of browsers by tracking the runtime behaviors of the browsers.

Many existing fuzzers [174, 177, 130, 10] focus on scheduling promising seeds, adopting dynamic analysis techniques or utilizing an additional constraint solver to enhance code coverage. In Chapter 5, we propose *MirageFuzz* to enhance the exploration capacity of each seed by reducing the program dependencies for conditional statements to reduce the difficulties of accessing their program states.

8.2 Studies on Fuzzing

Shen et al. [133] investigated different bugs on different deep learning compilers. Metzman et al. [99] introduced a platform for developers and researchers to evaluate different fuzzers. Although they studied *Havoc* associated with fuzzers, they did not evaluate it independently. Klees et al. [76] surveyed the recent research literature and assessed the experimental evaluations to illustrate the essential experimental setup for reliable experiments for fuzzing. We actually follow the instruction of this work to construct our initial seed corpus. Furthermore, Herrera et al. [60] systematically investigated and evaluated how seed selection affects the performance of a fuzzer to expose vulnerabilities in real-world systems. Many researchers studied the rationales behind fuzzing approaches. Liang et al. [91] presented the main obstacles and corresponding typical solutions for fuzzing. Tonder et al. [157] presented a technique to map crashing inputs to unique bugs using program transformation. Choi et al. [31] experimented on thousands of DLL files and API functions in Windows system to reveal the potential security issue in Windows. In this dissertation, we first conduct the extensive study on *Havoc* in Chapter 3 to demonstrate that *Havoc* is a powerful fuzzer, and have also shown that it is possible to further advance *Havoc*. Next, we conduct an empirical study to investigate the power and limitation of neural program-smoothing-based fuzzing and reveal various findings/guidelines for future learning-based fuzzing research in Chapter 4.

8.3 Compiler and JVM Testing

Researchers have spent large effort on compiler testing. To date, a number of techniques have been proposed to automatically generate programs for compiler testing. Yang et al. [163] proposed Csmith, which generates the seeding C programs to explore

C compiler while preventing the undefined and unspecified behaviors that might cause testing insufficiency. Reddy et al. [122] proposed RLCheck, which utilizes reinforcement learning to generate diverse valid inputs to explore the programs requiring strict validation on the inputs, e.g., JavaScript engine. Eberlein et al. [44] developed EVOGFUZZ, an evolutionary grammar-based fuzzing approach to optimize the probabilities to generate test inputs that are likely to trigger unexpected behaviors for applications with common input formats (JSON, JavaScript, or CSS3). For JVM testing, Yoshikawa et al. [166] proposed a random Java program generator based on the predefined syntax to expose JVM vulnerabilities by differential testing. Sirer et al. [38] proposed lava to generate Java programs for JVM testing via randomly iterating over the Java grammar productions. Boujarwah et al. [12] utilized the predefined grammar of Java programs to generate semantics-correct test cases to fuzz JVM. More recently, Zhao et al. [176] proposed JavaTailor to generate testing programs by learning information from historical bug-revealing test programs to expose JVM defects.

Compared with the above-mentioned generation-based testing approaches which either generate seeding programs from scratch (e.g., RLcheck [122]), or require program samples with additional efforts (e.g., JavaTailor [176]), the mutation-based testing approaches are usually launched with specifically designed mutators and collected seed corpus. To systematically parse existing real-world code for producing discrepancy-induced programs, Le et al. [82] introduced equivalence modulo inputs (EMI) to mutate seed programs and validate the quality of compilers. Zhang et al. [175] introduced the concepts of the skeletal program enumeration and replaced the variables in the original skeletal program to generate control flows for compiler testing. Sun et al. [142] tested compilers via mutating the live code of the input programs regardless the restriction of only mutating the unreachable regions of input programs. Donaldson et al. [40] developed GLFuzz for testing OpenGL shading language compilers based on semantics-preserving program transformations. Park et al. [114] proposed Die to fuzz the JavaScript engine via the aspect preservation mutators. In terms of testing JVM, Chen et al. [28] proposed classfuzz, which utilizes Markov Chain Monte Carlo [63, 4] to guide mutation via designed mutators. They also proposed classming [26] to leverage the power of manipulating the control flows of seeding class files to test the execution engine of JVM. Ouyang et al. [110] proposed MirrorTaint to track the data flows on JVM-based microservice applications for exposing missing data relations.

Compared with traditional compiler and JVM testing approaches, our proposed *JITfuzz* in Chapter 6 can be readily applied to test JITs thoroughly with the well-designed mutators and mutator scheduler under given collection of seed inputs. Moreover, we propose *SJFuzz* in Chapter 7 which adopts seed and mutator schedulers based on easy-to-catch runtime discrepancy/diversity information, i.e., acquiring no extra knowledge of bytecode constraints or JVM specifications comparing with existing approaches.

Chapter 9

Conclusion

9.1 Summary

In this dissertation, we first conduct two comprehensive studies to investigate representative fuzzing strategies, i.e., the *Havoc* mechanism and the gradient-based fuzzing strategies. Chapter 3 investigates the impact and design of a random fuzzing strategy *Havoc*. The evaluation results demonstrate that the pure *Havoc* can already achieve superior edge coverage and vulnerability detection compared with other fuzzers. Moreover, integrating *Havoc* to a fuzzer or extending total execution time for *Havoc* can also increase the edge coverage significantly. The performance gap among different fuzzers can also be considerably reduced by appending *Havoc*. At last, we also design a lightweight approach to further boost *Havoc* by dynamically adjusting its mutation strategy. Chapter 4 investigates the strengths and limitations of neural program-smoothing-based fuzzing approaches, e.g., *MTFuzz* and *Neuzz*. Our results also reveal various findings/guidelines for advancing future fuzzing research.

Next, we propose a coverage-guided fuzzer, namely *MirageFuzz*, to mitigate the dependencies between program branches when executing seeds for enhancing their exploration power on program states. Chapter 5 presents the detail of *MirageFuzz* which performs dual fuzzing for the original program and the phantom program simultaneously and adopts the taint-based mutation mechanism to generate new mutants by combining the resulting seeds from dual fuzzing via taint analysis. The evaluation results show that *MirageFuzz* outperforms the baseline fuzzers from 13.42% to 77.96% in terms of edge coverage averagely in our benchmark. Moreover, we have been inspired to adopt the findings out of these research work on typical foundational software systems with complicated constraints. Specifically for JVMs, our fuzzers [162, 161] successfully exposed 47 confirmed real-world JVM/JIT bugs. Chapter 5 and Chapter 6 present their details respectively.

9.2 Future Work

In pivotal domains such as chip and electronic device development, effective and efficient testing procedures are essential for ensuring their robustness. However, the existence of numerous intricate details and complex constraints in these critical areas poses significant challenges in effectively testing these applications. For instance, testing hardware devices often need to operate within limited computational resources. Many existing testing techniques such as symbolic execution require substantial computational resources, making their direct application in these domains challenging. Additionally, applications in these domains often operate under complex constraints, necessitating the design of effective testing strategies to thoroughly explore program states without compromising these constraints. Therefore, we will try to develop testing techniques in a limited resource testing scenario. Next, we intend to focus on developing more effective and efficient testing strategies for challenging fields regarding domestic foundational software systems such as CPU/GPU testing.

In the short term, we intend to improve the traditional testing techniques to activate them in a scenario with limited computational resources. One good start is to develop a general virtual device framework to simulate the feedback of a real device with simple configurations. By mapping a real device into a general virtual machine, we not only enable the use of testing techniques that demand greater computational resources but also facilitate the concurrent initiation of multiple testing processes, akin to testing traditional software. This approach thereby allows for increased testing parallelism, unrestricted by the actual quantity of physical devices.

In the long term, we are passionate about diving into more challenging domains such as CPU/GPU testing. For CPU/GPU development, testing efficiency is paramount. Developers need to constantly ascertain if each existing design within the system has any issues, as any flaw could potentially result in a rollback, significantly elongating the development cycle. Simultaneously, the original code in CPU/GPU development often remains opaque to testing tools, unlike code instrumentation feasible in fuzz testing, making efficient testing of CPU/GPU applications a substantial challenge. One direction is to model the functionalities of CPUs/GPUs, defining and designing testing oracles and inputs that activate corresponding functionalities, and leveraging existing knowledge to generate diverse inputs to comprehensively test various functions of CPUs/GPUs.

Appendix A

About Appendix

The appendix is usually used to provide some supplementary materials for the publications. For example, some experimental results, network architecture, detailed experimental settings or proving of the theories. You can have more than one appendices to provide the materials for different uses.

Appendix B

Appendix Title Here

Write your Appendix content here.

Appendix C

Appendix Title Here

Write your Appendix content here.

Bibliography

[1] *SJFuzz's source code*. https://github.com/lochnagarr/JITFuzz. 2022.

[2] *All experiments detail in the paper*. https://github.com/WorldExecute/exprs. 2022.

[3] *An example for Depth-Ensured transition*. https://github.com/lochnagarr/JITFuzz/blob/main/examples/Depth-Ensured/NumberUtils.java.

[4] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan. "An introduction to MCMC for machine learning". In: *Machine learning* 50.1 (2003), pp. 5–43.

[5] *Apache Project*. https://projects.apache.org/projects.html?language. 2022.

[6] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia. "rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes". In: *SoftwareX* 12 (2020), p. 100635. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100635. URL: https://www.sciencedirect.com/science/article/pii/S2352711020303484.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2 (2002), pp. 235–256.

[8] I. Bartolini, P. Ciaccia, and M. Patella. "String matching with metric trees using an approximate distance". In: *String Processing and Information Retrieval: 9th International Symposium, SPIRE 2002 Lisbon, Portugal, September 11–13, 2002 Proceedings 9*. Springer. 2002, pp. 271–283.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[11] M. Böhme, V.-T. Pham, and A. Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for

Computing Machinery, 2016, pp. 1032–1043. ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[12] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. "Testing syntax and semantic coverage of Java language compilers". In: *Information and Software Technology* 41.1 (1999), pp. 15–28.

[13] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. "Program transformation with scoped dynamic rewrite rules". In: *Fundamenta Informaticae* 69.1-2 (2006), pp. 123–178.

[14] C. E. Brown. "Coefficient of variation". In: *Applied multivariate statistics in geohydrology and related sciences*. Springer, 1998, pp. 155–157.

[15] C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[16] A. Calleja, J. Tapiador, and J. Caballero. "The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development". In: *IEEE Transactions on Information Forensics and Security* 14.12 (2019), pp. 3175–3190. DOI: 10.1109/TIFS.2018.2885512.

[17] S. Chaudhuri and A. Solar-Lezama. "Smoothing a Program Soundly and Robustly". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 277–292. DOI: 10.1007/978-3-642-22110-1_22. URL: https://doi.org/10.1007/978-3-642-22110-1%5C_22.

[18] S. Chaudhuri and A. Solar-lezama. "Smooth interpretation". In: *In PLDI*. 2010.

[19] J. Chen, J. Wang, C. Song, and H. Yin. "JIGSAW: Efficient and Scalable Path Constraints Fuzzing". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 18–35. DOI: 10.1109/SP46214.2022.9833796.

[20] J. Chen, H. Ma, and L. Zhang. "Enhanced compiler bug isolation via memoized search". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020, pp. 78–89. DOI: 10.1145/3324884.3416570.

[21] P. Chen and H. Chen. "Angora: Efficient fuzzing by principled search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[22] P. Chen, J. Liu, and H. Chen. "Matryoshka: fuzzing deeply nested branches". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 499–513. DOI: 10.1145/3319535.3363225.

[23] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. "Taming Compiler Fuzzers". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 197–208. ISSN: 0362-1340. DOI: 10.1145/2499370.2462173. URL: https://doi.org/10.1145/2499370.2462173.

[24] Y. Chen, M. Ahmadi, B. Wang, L. Lu, et al. "{MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 2020, pp. 77–92.

[25] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. "Savior: Towards bug-driven hybrid testing". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1580–1596.

[26] Y. Chen, T. Su, and Z. Su. "Deep differential testing of JVM implementations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1257–1268.

[27] Y. Chen, T. Su, and Z. Su. "Deep differential testing of JVM implementations". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 1257–1268.

[28] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. "Coverage-directed differential testing of JVM implementations". In: *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[29] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[30] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. "Escape Analysis for Java". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '99. Denver, Colorado, USA: Association for Computing Machinery, 1999, pp. 1–19. ISBN: 1581132387. DOI: 10.1145/320384.320386. URL: https://doi.org/10.1145/320384.320386.

[31] Y. Choi, H. Kim, and D. Lee. "An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing". In: *2008 10th International Conference on Advanced Communication Technology*. Vol. 2. 2008, pp. 1473–1475. DOI: 10.1109/ICACT.2008.4494042.

[32] *Coevolutionary Algorithm*. https://wiki.ece.cmu.edu/ddl/index.php/Coevolutionary_algorithms. 2022.

[33] *Complement (set theory)*. https://en.wikipedia.org/wiki/Complement_(set_theory). 2023.

[34] *Control-flow graph generating pass of LLVM*. https://llvm.org/docs/Passes.html#dot-cfg-print-cfg-of-function-to-dot-file. 2022.

[35] L. De Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[36] *Definition of Distinct Discrepancy*. https://github.com/fuzzy000/SJFuzz/blob/main/src/com/djfuzz/solver/JVMOutputParser.java. 2022.

[37] J. DeMott, R. Enbody, and W. F. Punch. "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing". In: *BlackHat and Defcon* (2007).

[38] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[39] *Dominator and Immediate dominator, WiKipedia*. https://en.wikipedia.org/wiki/Dominator_(graph_theory). 2022.

[40] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. "Automated testing of graphics shader compilers". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–29.

[41] *DragonWell11*. https://github.com/alibaba/dragonwell11. 2022.

[42] *DragonWell8*. https://github.com/alibaba/dragonwell8. 2022.

[43] Z. Du, Y. Li, Y. Liu, B. Mao, L. Chen, J. Guo, Z. He, D. Mu, C. Pang, R. Yu, et al. "WindRanger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks". In: *2022 IEEE/ACM 44st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2022. DOI: 10.1145/3510003.3510197.

[44] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske. "Evolutionary Grammar-Based Fuzzing". In: *Search-Based Software Engineering*. Ed. by A. Aleti and A. Panichella. Cham: Springer International Publishing, 2020, pp. 105–120. ISBN: 978-3-030-59762-7.

[45] J. L. Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211.

[46] S. Embleton, S. Sparks, and R. Cunningham. "Sidewinder: An Evolutionary Guidance System for Malicious Input Crafting". In: *Black Hat USA* (2006).

[47] *Escape*. https://en.wikipedia.org/wiki/Escape_analysis. 2022.

[48] R. B. Evans and A. Savoia. "Differential Testing: A New Approach to Change Detection". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 549–552. ISBN: 9781595938114. DOI: 10.1145/1287624.1287707. URL: https://doi.org/10.1145/1287624.1287707.

[49] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++: Combining incremental steps of fuzzing research". In: *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*. 2020.

[50] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan. "{BrakTooth}: Causing Havoc on Bluetooth Link Manager via Directed Fuzzing". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1025–1042. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/garbelini.

[51] *Gestalt Pattern Matching Distance*. https://en.wikipedia.org/wiki/Gestalt_pattern_matching. 2022.

[52] github. *GitHub*. 2022. URL: https://github.com/.

[53] *Github Repository. 2022. MirageFuzz*. https://github.com/WorldExecute/fuzzer. 2022.

[54] *Global Escape*. https://wiki.openjdk.org/display/HotSpot/EscapeAnalysis. 2022.

[55] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed automated random testing". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[56] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. "Automated whitebox fuzz testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[57] A. Graves and J. Schmidhuber. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". In: *Neural networks* 18.5-6 (2005), pp. 602–610.

[58] Z. Gui, H. Shu, F. Kang, and X. Xiong. "FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution". In: *IEEE Access* 8 (2020), pp. 29826–29841. DOI: 10.1109/ACCESS.2020.2973043.

[59] S. Hashima, M. M. Fouda, Z. M. Fadlullah, E. M. Mohamed, and K. Hatano. "Improved UCB-based Energy-Efficient Channel Selection in Hybrid-Band Wireless Communication". In: *2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2021, pp. 1–6.

[60] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and T. Hosking. "Seed Selection for Successful Fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual Event, USA, 2021.

[61] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[62] *How the JIT compiler optimizes code*. https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code. 2022.

[63] D. Huang, J.-B. Tristan, and G. Morrisett. "Compiling Markov chain Monte Carlo algorithms for probabilistic modeling". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 111–125.

[64] H. Huang, P. Yao, R. wu, Q. Shi, and C. Zhang. "Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction". In: May 2020, pp. 1613–1627. DOI: 10.1109/SP40000.2020.00063.

[65] *Inlining*. https://en.wikipedia.org/wiki/Inline_expansion. 2022.

[66] *J9*. http://www.ibm.com/developerworks/java/jdk. 2022.

[67] *Jaro Distance*. https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance. 2022.

[68] *JDK-8280126*. https://bugs.openjdk.org/browse/JDK-8280126. 2022.

[69] *jhead use-of-uninitialized-value bug issue*. https://github.com/Matthias-Wandel/jhead/issues/53. 2022.

[70] *JIt Bug*. https://github.com/eclipse/openj9/issues/9381. 2020.

[71] *JRockit*. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/webdocs/index.html. 2022.

[72] *JUnit Official Website*. https://junit.org/. 2022.

[73] *Just-in-time compilation*. https://en.wikipedia.org/wiki/Just-in-time_compilation. 2021.

[74] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. "libdft: Practical dynamic data flow tracking for commodity systems". In: *Proceedings of the 8th ACM*

SIGPLAN/SIGOPS conference on Virtual Execution Environments. 2012, pp. 121–132. DOI: 10.1145/2365864.2151042.

[75] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95-international conference on neural networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[76] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. "Evaluating fuzz testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[77] J. Kukucka, L. Pina, P. Ammann, and J. Bell. "CONFETTI: Amplifying Concolic Guidance for Fuzzers". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 438–450. DOI: 10.1145/3510003.3510628.

[78] A. Kuleshov, P. Trifanov, V. Frolov, and G. Liang. "Diktat: Lightweight Static Analysis for Kotlin". In: *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2021, pp. 365–370.

[79] *laf-intel instrumentation*. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.laf-intel.md. 2022.

[80] C. Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD conference*. Vol. 5. 2008, pp. 1–20.

[81] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[82] V. Le, M. Afshari, and Z. Su. "Compiler Validation via Equivalence modulo Inputs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 216–226. ISBN: 9781450327848. DOI: 10.1145/2594291.2594334. URL: https://doi.org/10.1145/2594291.2594334.

[83] V. Le, C. Sun, and Z. Su. "Finding deep compiler bugs via guided stochastic program mutation". In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 386–399. DOI: 10.1145/2858965.2814319.

[84] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[85] C. Lemieux. *Comments on Havoc*. https://twitter.com/cestlemieux/status/1524438583184138240. 2022.

[86] C. Lemieux and K. Sen. "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 475–485.

[87] V. I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[88] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. "Steelix: program-state based binary fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on*

Foundations of Software Engineering. 2017, pp. 627–637. DOI: 10.1145/3106237.3106295.

[89] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu. "V-fuzz: Vulnerability-oriented evolutionary fuzzing". In: *arXiv preprint arXiv:1901.01142* (2019).

[90] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. "PATA: Fuzzing with Path Aware Taint Analysis". In: *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 154–170. DOI: 10.1109/SP46214.2022.00010. URL: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00010.

[91] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. "Fuzz testing in practice: Obstacles and solutions". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 562–566. DOI: 10.1109/SANER.2018.8330260.

[92] *libpng - library for use in applications that read, create, and manipulate PNG*. https://github.com/glennrp/libpng. 2022.

[93] *List of JVM languages, WiKipedia*. https://en.wikipedia.org/wiki/List_of_JVM_languages. 2022.

[94] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. "{MOPT}: Optimized mutation scheduling for fuzzers". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 1949–1966.

[95] *Main Repo for SJFuzz*. https://github.com/fuzzy000/SJFuzz. 2022.

[96] Matthias-Wandel. *The jhead Repo*. https://github.com/Matthias-Wandel/jhead. 2021.

[97] T. McCabe. "A Complexity Measure". In: vol. SE-2. 4. 1976, pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[98] *Memory SSA in LLVM*. https://llvm.org/docs/MemorySSA.html. 2022.

[99] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. "Fuzzbench: an open fuzzer benchmarking platform and service". In: *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 2021, pp. 1393–1403. DOI: 10.1145/3468264.3473932.

[100] *Monte Carlo Method*. https://en.wikipedia.org/wiki/Monte_Carlo_method. 2022.

[101] *Motivate Examples*. https://github.com/eclipse/openj9/issues/11683. 2021.

[102] *Motivate Examples*. https://github.com/eclipse/openj9/issues/11684. 2021.

[103] D. Novillo et al. "Memory SSA-a unified approach for sparsely representing memory operations". In: *Proceedings of the GCC Developers' Summit*. Citeseer. 2007, pp. 97–110.

[104] *OpenJ9*. https://www.eclipse.org/openj9/. 2022.

[105] *OpenJ9 assertion failure*. https://github.com/eclipse-openj9/openj9/issues/15639. 2022.

[106] *OpenJ9 Optimizer Vulnerability*. https://github.com/eclipse-openj9/openj9/issues/15764. 2022.

[107] *OpenJDK*. https://jdk.java.net/. 2022.

[108] *OpenJDK JDK 19 Release-Candidate Builds*. https://jdk.java.net/19/. 2022.

[109] *Oraclejdk*. https://www.oracle.com/java/technologies/downloads/. 2022.

[110] Y. Ouyang, K. Shao, K. Chen, R. Shen, C. Chen, M. Xu, Y. Zhang, and L. Zhang. "MirrorTaint: Practical Non-intrusive Dynamic Taint Tracking for JVM-based Microservice Systems". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Association for Computing Machinery, 2023.

[111] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. "Semantic fuzzing with zest". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 329–340.

[112] M. Paleczny, C. Vick, and C. Click. "The Java HotSpot™ Server Compiler". In: *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. Monterey, CA: USENIX Association, Apr. 2001.

[113] M. Paleczny, C. Vick, and C. Click. "The Java HotspotTM Server Compiler". In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, 2001, p. 1.

[114] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. "Fuzzing JavaScript Engines with Aspect-preserving Mutation". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1629–1642. DOI: 10.1109/SP40000.2020.00067.

[115] *PCRE2 - Perl-Compatible Regular Expressions*. https://github.com/PCRE2Project/pcre2. 2022.

[116] *pcre2 infinite loop bug issue*. https://github.com/PCRE2Project/pcre2/issues/141. 2022.

[117] *Perl - a highly capable, feature-rich programming language*. https://www.perl.org/. 2022.

[118] *pngfix use-of-uninitialized-value bug issue*. https://github.com/glennrp/libpng/issues/424. 2022.

[119] R. Potharaju and N. Jain. "Demystifying the dark side of the middle: A field study of middlebox failures in datacenters". In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013, pp. 9–22.

[120] E. J. Powley, D. Whitehouse, and P. I. Cowling. "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search". In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE. 2013, pp. 1–8.

[121] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *NDSS*. Vol. 17. 2017, pp. 1–14.

[122] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. "Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 1410–1421.

[123] Y. Ren, S. Krishnamurthi, and K. Fisler. "What Help Do Students Seek in TA Office Hours". In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. Association for Computing Machinery. 2019, pp. 41–49.

[124] G. Repository. *Havoc-Study*. https://github.com/MagicHavoc/Havoc-Study. 2021.

[125] G. Repository. *Program smoothing fuzzing*. https://github.com/PoShaung/program-smoothing-fuzzing. 2021.

[126] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[127] E. J. Schwartz, T. Avgerinos, and D. Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[128] D. She. *Comments on ML-based fuzzing*. https://twitter.com/DongdongShe/status/1732271632675447063. 2023.

[129] D. She. *neuzz repository*. https://github.com/Dongdongshe/neuzz. 2020.

[130] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray. "MTFuzz: fuzzing with a multi-task neural network". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 737–749.

[131] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. "NEUZZ: Efficient fuzzing with neural program smoothing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 803–817.

[132] D. She, A. Shah, and S. Jana. "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 2194–2211. DOI: 10.1109/SP46214.2022.9833761.

[133] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen. "A comprehensive study of deep learning compiler bugs". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 968–980.

[134] Z. Shen, R. Roongta, and B. Dolan-Gavitt. "Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds". In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Aug. 2022, pp. 1275–1290. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/shen-zekun.

[135] C. Shou, I. B. Kadron, Q. Su, and T. Bultan. "Corbfuzz: Checking browser security policies with fuzzing". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 215–226. DOI: 10.1109/ASE51524.2021.9678636.

[136] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[137] *Simplification*. https://en.wikipedia.org/wiki/Computer_algebra#Simplification. 2022.

[138] *Spring Verify Error*. https://github.com/eclipse/openj9/issues/5676. 2020.

[139] E. Stepanov and K. Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 46–55. DOI: 10.1109/CGO.2015.7054186.

[140] *strip out-of-memory bug issue*. https://sourceware.org/bugzilla/show_bug.cgi?id=29495. 2022.

[141] *Structured Locking Issue*. https://github.com/eclipse/openj9/issues/9276. 2020.

[142] C. Sun, V. Le, and Z. Su. "Finding Compiler Bugs via Live Code Mutation". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 849–863. ISSN: 0362-1340. DOI: 10.1145/3022671.2984038. URL: https://doi.org/10.1145/3022671.2984038.

[143] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[144] Q. Tao, W. Wu, C. Zhao, and W. Shen. "An automatic testing approach for compiler based on metamorphic testing technique". In: *2010 Asia Pacific Software Engineering Conference*. IEEE. 2010, pp. 270–279.

[145] *The Java Virtual Machine Specification*. https://docs.oracle.com/javase/specs/index.html. 2022.

[146] R. Vallee-Rai and L. J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

[147] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot: A Java bytecode optimization framework". In: *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[148] *Verify Bug*. https://github.com/eclipse/openj9/issues/9385. 2020.

[149] X. Wang, J. Tang, M. Yu, G. Yin, and J. Li. "A UCB1-Based Online Job Dispatcher for Heterogeneous Mobile Edge Computing System". In: *2018 Third International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC)*. IEEE. 2018, pp. 1–6.

[150] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization." In: *NDSS*. 2020.

[151] M. Weiser. "Program slicing". In: *IEEE Transactions on software engineering* 4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[152] Wikipedia. *Exposing Bugs by Fuzzing*. https://en.wikipedia.org/wiki/Fuzzing. 2021.

[153] Wikipedia. *Fuzzing*. en.wikipedia.org/wiki/Fuzzing. Online; accessed 27-Jan-2020. 2020.

[154] Wikipedia. *Jaccard Distance*. https://en.wikipedia.org/wiki/Jaccard_index. 2021.

[155] Wikipedia. *Multi-armed Bandit Problem*. https://en.wikipedia.org/wiki/Multi-armed_bandit. 2021.

[156] Wikipedia. *Socket programming*. https://en.wikipedia.org/wiki/Network_socket. 2021.

[157] R. Wilhelm, H. Seidl, and S. Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

[158] M. Wu, K. Chen, Q. Luo, J. Xiang, J. Qi, J. Chen, H. Cui, and Y. Zhang. "Enhancing Coverage-Guided Fuzzing via Phantom Program". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1037–1049.

[159] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang. "One Fuzzing Strategy to Rule Them All". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022.

[160] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang. "Evaluating and Improving Neural Program-Smoothing-based Fuzzing". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 847–858. DOI: 10.1145/3510003.3510089.

[161] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang. "JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 56–68. DOI: 10.1109/ICSE48619.2023.00017.

[162] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, G. Yang, and Y. Zhang. "SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023, pp. 1062–1074.

[163] X. Yang, Y. Chen, E. Eide, and J. Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 283–294. ISBN: 9781450306638. DOI: 10.1145/1993498.1993532. URL: https://doi.org/10.1145/1993498.1993532.

[164] Y. Yang, B. Jenny, T. Dwyer, K. Marriott, H. Chen, and M. Cordeil. "Maps and globes in virtual reality". In: *Computer Graphics Forum*. Vol. 37. 3. Wiley Online Library. 2018, pp. 427–438.

[165] X. Yao, Y. Liu, and G. Lin. "Evolutionary programming made faster". In: *IEEE Transactions on Evolutionary computation* 3.2 (1999), pp. 82–102.

[166] T. Yoshikawa, K. Shimura, and T. Ozawa. "Random program generator for Java JIT compiler test system". In: *Third International Conference on Quality Software, 2003. Proceedings.* 2003, pp. 20–23. DOI: 10.1109/QSIC.2003.1319081.

[167] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 745–761.

[168] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 745–761.

[169] M. Zalewski. *Edge Coverage Dopted in AFL*. https://groups.google.com/g/afl-users/c/fOPeb62FZUg/m/LYxgPYheDwAJ. 2016.

[170] M. Zalewski. *AFL Official Seed Corpus*. http://lcamtuf.coredump.cx/afl/demo/. 2021.

[171] M. Zalewski. *American Fuzz Lop*. https://github.com/google/AFL. 2020.

[172] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu. "Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing". In: *Network and Distributed Systems Security (NDSS) Symposium 2022*. 2022. DOI: https://dx.doi.org/10.14722/ndss.2022.24314.

[173] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen. "Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 1658–1668. DOI: 10.1145/3510003.3510063.

[174] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. "DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 132–142. DOI: 10.1145/3238147.3238187.

[175] Q. Zhang, C. Sun, and Z. Su. "Skeletal Program Enumeration for Rigorous Compiler Testing". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 347–361. ISSN: 0362-1340. DOI: 10.1145/3140587.3062379. URL: https://doi.org/10.1145/3140587.3062379.

[176] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang. "History-Driven Test Program Synthesis for JVM Testing". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1133–1144. ISBN: 9781450392211. DOI: 10.1145/3510003.3510059. URL: https://doi.org/10.1145/3510003.3510059.

[177] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu. "DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020, pp. 347–358.